



X86 64-Bit Extension Multimedia Instruction Set Architecture

Motivation
Objectives & Method
Data Types
Instruction Set
Competitive Analysis
Open Issues

MM

iMultimedia

EXHIBIT NO. 502
W. Michael
9/22/85

A. Peleg - 4/92

64-bit Multimedia ISA Ratification Summit

Intel Confidential

1 of 21

Motivation

Boost performance of CSC applications. Possible because many CSC applications have following characteristics:

- Small native data elements.
- Regular and predictable memory access patterns.
- Relatively compute intensive time consuming inner loops.
- Reoccurring localized operations performed over data.
- Control flow in many cases is data independent.

Applications considered are: 2D-3D graphics, Image processing, Video compression/decompression, capture and playback(MPEG1+2), Teleconferencing (Px64), Audio, Lossless compressions (Networking), Recognition processes (Speech, Hand-writing).

A. Peleg - 4/92

Intel Confidential

2 of 21

64-bit Multimedia ISA Ratification Summit

Objectives

- Boost performance of PC & MSCG current and future Multimedia applications via new ISA on general purpose CPU.
- Be consistent with X86 64-Bit Extension core ISA.
- Make Multimedia ISA capabilities accessible to compilers.

Method

- Add new vector data types and general/simple instructions to manipulate them.
- 3 Tier SW support:
 - C code vectorization.
 - C++ encapsulation.
 - Assembler coded libraries & assembler macros.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

3 of 21

Data Types

- Pack signed/unsigned fixed Point data elements into a 64-bit register:
- Byte Vector (8 x 8 bit), Word Vector (4 x 16 bit)
- Mappable to common data types in Multimedia applications: Pixels, Fixed point coefficients (dct, filter), Characters, Integer Coordinates (2D, 3D).

Byte Vector (8 x 8 bits): (b) suffix.

6 3	5 8	5 8	4 8	4 8	4 0	3 9	3 2	3 1	2 4	2 3	1 6	1 5	0 8	0 7	0 0

Word Vector (4 x 16 bits): (w) suffix.

6 3	4 8	4 8	3 2	3 1	1 6	1 5	0 8

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

4 of 21

Instruction Set

Standard Arithmetic Primitives (Vector operation)

- Add, Sub, Mul, Cmp, Shift

Multimedia Specific Operations (Vector operation)

- Conversions between fixed point data types.
- Merging of source operands.

Saturating mode

- Add/subtract/pack instructions support saturating mode (clip to max/min values).

Note: No immediates available in Multimedia ISA. Constants prepared in memory before hand (in replicated format) and loaded into registers, or values replicated on the fly.

Suggestion: Change prefix of Multimedia ISA from "X" (extension of extension) to "V" (Vector).

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

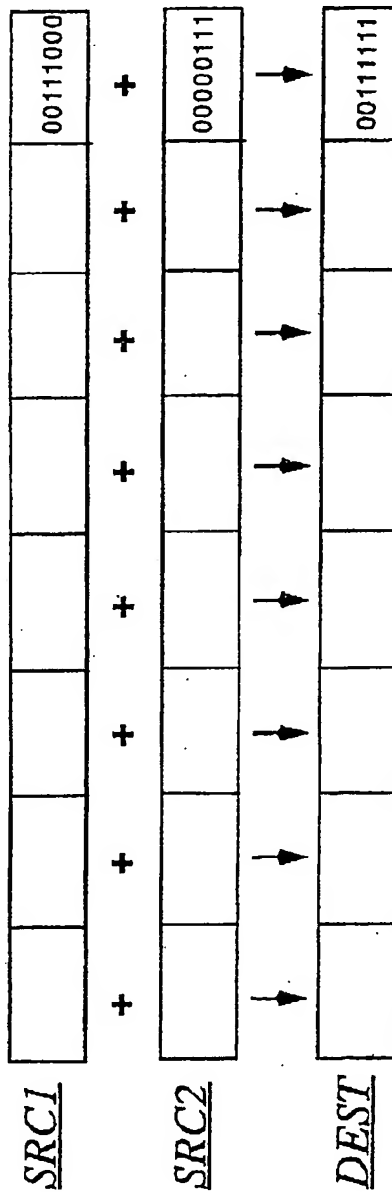
5 of 21

Instruction Set (cont.)

Fixed Point (signed/unsigned) Addition/Subtraction

xiaddp/xaddp src1, src2, destxisubp/xsubp src1.src2.dest

Example: xaddb



p suffix - one of b or w

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

6 of 21

Instruction Set (cont.)

Fixed Point Multiply

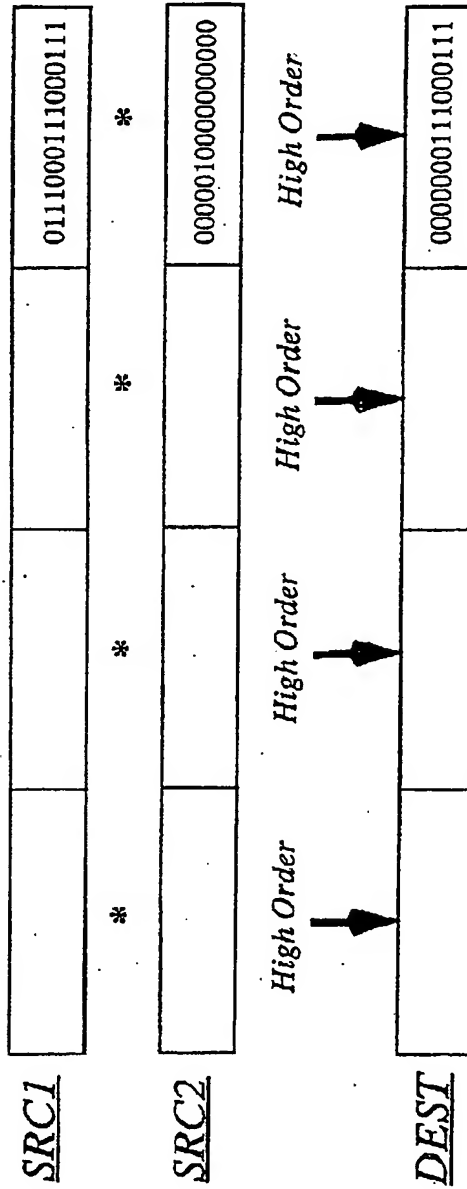
Used for multiplying by fractional matrix coefficients.

simulw

src1, src2, dest

- Result taken from high order bits of result (integer part of result).

Example: simulw



p suffix - one of b or w

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

7 of 21

Instruction Set (cont.)

Comparison

xcmpccp src1, src2, dest

- cc can be one of: e (equal), ne (not equal), g/gu (greater than signed/unsigned), ge/geu (greater than or equal signed/unsigned). No need for 'less than' because of symmetry.
- If result of element comparison true, dest element all ones. Else all zero's.

Example: xcmpew

SRC1

		0000000000001111	0111000111000111
--	--	------------------	------------------

==

==

==

==

SRC2

		0111000111000111	0111000111000111
--	--	------------------	------------------

True

False

False

True

DEST

1111111111111111	0000000000000000	0000000000000000	1111111111111111
------------------	------------------	------------------	------------------

p suffix - one of b or w

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

8 of 21

Instruction Set (cont.)

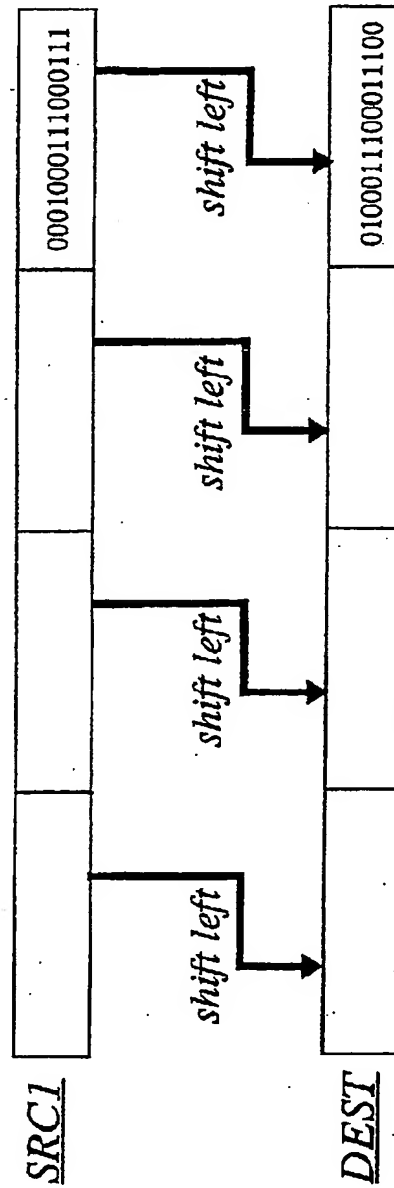
Left/Right Arithmetic Shift

Shift instructions will be typically used for scaling of fixed point data (in addition to standard shift & mask operations for bit field extraction).

xshlp src1, shift_count, dest

xsarp src1, shift_count, dest

Example: xshlw



p suffix - one of b or w

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

9 of 21

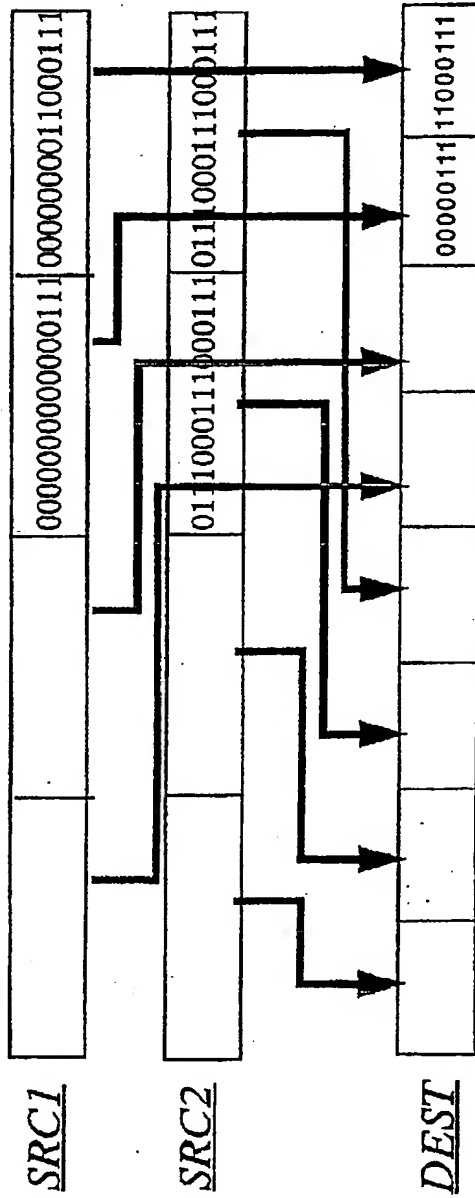
Instruction Set (cont.)

Packing of fixed point data types

xipackwb/xpackpwb *src1, src2, dest*

- Provide packing abilities to fixed point formats. Low order bits of src elements are moved to destination.
- Saturation is always active.

Example: *xpackwb*



A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

10 of 21

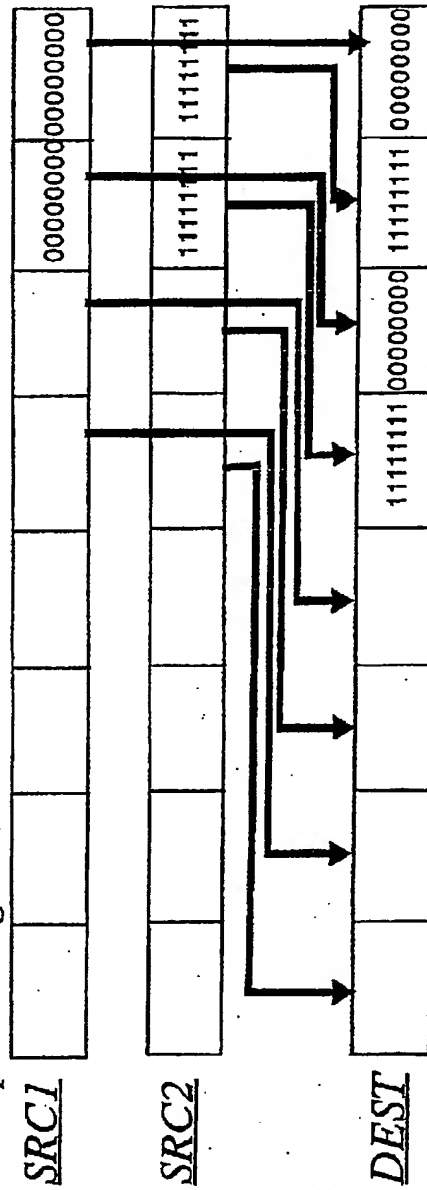
Source Merge

Typically used for planar \leftrightarrow chunky representation conversions and interpolated pixel combining.

xmergep

- Interleaved merge of two sources. Low order elements taken.

Example: xmergeb



p suffix - one of borrow

64-bit Multimedia ISA Ratification Summit

Intel Confidential

A. Peleg - 4/92

11 of 21



Competitive Analysis (General Purpose CPUs)

Table 1: Competitive Analysis

Architecture	2D/ Misalignment support	3D support	Imaging	More ... *
DEC Alpha - 21064	YES	NO	NO	NO
IBM RS6000	YES	NO	NO	NO
MIPS R4000	YES	NO	NO	NO
Motorola 88110	YES	YES	YES	NO
Intel 860	YES	YES	NO	NO
X86 64 bit	YES	YES	YES	YES

* - Full image processing capabilities, Audio, Video capture/playback/compression/decompression, Teleconferencing, Recognition processes.

A. Peleg - 4/92

Intel Confidential

12 of 21

64-bit Multimedia ISA Ratification Summit



Competitive Analysis

DEC Alpha- 21064- No advertized CSC/MM support - have 2D alignment support.

Relevant Data types - Implemented on INTEGER side.

Byte vector in 64 bit register - targeted at string operations.

Instructions - all are 2 source one destination.

Ldq_u/Stq_u -	align address before load/store (Bitblit).
Extract/Insert/Mask -	shift (byte count) and ext/ins/mask byte, word
Cmpbge -	Byte compare - result in bit per byte mask
Zap/Zapnot -	Conditional byte move.
Fetch/Fetch_m -	Large block prefetch to next mem Level- hint.

* - Most features were actually added to support byte and word accesses.

A. Peleg - 4/92

Intel Confidential

13 of 21

64-bit Multimedia ISA Ratification Summit



Competitive Analysis

IBM RS6000 - No advertized CSC/MM support - have 2D misalign support.

Relevant Data types - None.

Instructions - Have implicit additional sources.

Sleq/Sliq -	shift src1 merge with src2 control by mask (bitblit).
Ld/St -	w/o alignment checking - Auto increment.
Ldi/Stsi/Lm -	Load/store multiple bytes (up to 120).
Maskg/Maskir -	Generate mask (bitblit end cases).
Fms/Fma -	Floating point multiply - accumulate(2D/3D).
Lscbx -	Load string of bytes until match (string operations).
Doz/Dozi -	Difference or zero - saturation.

* - Multiple bytes operations not included in 'Power PC'.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

14 of 21



Competitive Analysis

MIPS R4000 - No advertized CSC/MM support - have 2D misalign support.

Relevant Data types - None.

Instructions -

Lwr/Lwl/Swl/Swr - start loading or storing from/to any misaligned address
Cplx - 4 Co-processor classes supported via special load/store instructions.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

15 of 21

Competitive Analysis

Motorola 88110 - Targeted mainly at 3D and Image Compositing.

Relevant Data types - Implemented on INTEGER side.

FP single, double and extended.

Fixed point packed - 4, 8, 16, 32 bits in 64 bit amount (paired 32 bit regs).

Fixed point types are: 4.0 unsign, 8.0 sign/unsign, 8.8 sign/unsign, 8.24 sign/unsign. Decimal point is conceptual, thus, any other type is possible.

Instructions- all are 2 source (2 paired registers) one destination (paired).

Pmul -

64x8->64 - Actually use Integer multiply.

Padd[s].[x].t - add/add with saturate. t - size of element, x - sign/unsign.

Psub[s].[x].t - subtract/subtract with saturate. Control in opcode.

Pcmpcc - 2x32 bit int or FP compare. Result 8 bit condition code.

Ppack.r.t - Pack from one data type to another. Shift src1 left and pack into vacant low order bytes. Xpack double throughput.

Punpk.t -

Unpack to one size up - sparse with zero's = xmerge.

Prot -

Rotate paired regs to modulo 4. Shift double support.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

16 of 21

Competitive Analysis

Intel 80860XP - Targeted only at 3D.

Relevant Data types - Implemented on FP side (Memory BW)

FP single, double.

Fixed point packed - 8, 16, 32 bits in 64 bit amount (paired 32 bit FP regs).

Fixed point types are: 8.0 unsign, 6.10 unsign, 8.24 unsign.

Instructions- have implicit sources and destinations..

Dual FP Operation - multiply - accumulate.

[p]fiadd.w - Long-integer add.

[p]fisub.w - Long-integer subtract.

[p]fzchks/1 - 16/32 Z buffer check. Accumulative in Pixel Mask.

[p]faddp - Add with pixel merge. Shift merge and write integer part.

[p]faddz - Add with Z merge. Shift merge and write integer part.

[p]form - Or with merge. Combine merge with fsrcl into fdest.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

17 of 21



Competitive Analysis

Conclusions

- Most Architectures have SW support for 2D alignment only - even if that was not the main purpose.
- Only Few General Purpose Architectures with Specific Graphics/MM ISA targeted mainly at 3D market segment.
- Graphics/MM ISA specific to one or two algorithms.
- Graphics/MM ISA not compiler visible.

Our MM ISA

- We have opportunity to differentiate and address the full extent of the emerging CSC/MM market.
- General Generic data types and operations.
- Extensive SW support planned: C++ encapsulation, C vectorization, ASM Macros + Libraries.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

18 of 21

P67 MM POR HW Implementation Complexity

Table 1: MM POR ISA HW Implementation

MM POR ISA	Unit	Needed HW	Frequency Effect	Area Effect	Effort
X[i]add, X[i]sub	Int ALU	Muxes to break Carry Saturate muxes Control for above	0.7->0.8clk int 0.8clk graphics	0.6K tr = 4% of one ALU	Small
Xcmpcc	Int ALU	Use muxes of int cmp Control for byte masks	None	1.5K tr = 10% of one ALU	Small
Xshl, Xsar	Int Shifter	Muxes to pad results with 0/1 instead of src	0.5->0.6clk int 0.7clk graphics	0.5K tr = 10% one shifter	Small
Ximulw	Int Kunit	Implement multiplier with 16 bit arrays. If ximul omitted - use P5 multiply for int +fp	2 cyc - Critical	>100K tr	Big
Xmerge + X[i]pack	New FUB	4->1 mux for src bytes OR gates for xpack saturation. 3->1 saturate muxes. Latches drivers (unit).	less 0.5clk	10K tr	Medium

Overall Effort: 3 + 1/2 Persons for the duration of the project.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

19 of 21

P67 MM Open Issues HW Implementation Complexity

Table 1: ISA Open Issues HW Implementation

ISA Open Issues	Unit	Needed HW	Frequency Eff'	Area Effect	Effort
8x8 Ximul	Int Kunit	Implement multiplier with 8 bit arrays. Ximulw implemented upon 8 bit arrays.	3 cyc - Critical	>120K tr	Big
Rounding on Ximul	Int Kunit	Logic on Low order bits to determine rounding. Incrementer for round.	Critical	Large Area for rounding logic and incrementers	Big
Rounding on Xsar	Int Shifter	See above.	Critical	See Above	Big
Xdist (Absolute difference)	Int ALU ?	Perform xisub. check if negative. complement result. Increment result. Mux output.	Critical	5K tr = 30% of one ALU	Big
Saturating Xshl	Int Shifter	Or/Nand on all possible shifted out bit amounts. Unify Cases. 5->1 saturate out mux.	0.7clk->0.95clk - Critical	2.5K tr = 45% of shifter	Big
Conditional Store	Dunit Bunit	Add two 8 bit latches miss handle unit, L2CQ and bus CQ.	None	2.5K tr + inter-connect	Medium

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

20 of 21

Table 1: MM POR ISA HW Implementation

MM POR ISA	Unit	Needed HW	Frequency Effect	Area Effect	Effort
Vector mul Element	Int Kunit	Another state in Kunit FSM. Input Muxing.	Adding Criticality	Negligible	Medium
Xshr - Logical shr	Int Shifter	Nearly none.	None	None	Small
Low order result ximul	Int Kunit	2->1 mux on output	None	Negligible	Small
Xmin/Xmax	Int ALU	4->1 output mux	None	None	Small
Xabsoute	Int ALU	Control ALU by sign	None	None	Small
Pack from high byte	New FUB	2->1 mux	None	Negligible	Small
Xreplicate	New FUB	More muxing	0.5->0.6clk	20%	Small
Xaverage	Int ALU	Enlarge result mux	None	Negligible	Small
Xcmp give bit mask	Int ALU	Simplify output stage	None	None	Small
Saturation Control in Opcode	Int ALU	Opcode space only	None	None	None

Open Issues probable to be added - from small effort list.

Overall Effort beyond MM POR: 1/4 - 1/2 person for duration of project.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

21 of 21

Customer Inputs

Table 1: Collection of customer requests and comments

Company	Vector Instructions	Data types	Saturation needs	Conditional Select	Serial Bit operation
NCR Multimedia	*	8, 16, 24		*	
Pixar - 3D Image	Maybe find usage	unsign 8, sign 16	8 unsign		
Dragon - speech	*	sign 8, unsign 16	8 unsign	*	
Adobe - Image	*	2.14, 2.30	16 sign		
Mediavision - Video	*	sign 8	sign 8, unsign 16	*	*
Micrografx - MM	*	16.16	*	*	
Xing - Video	*	unsign 3.13	*		*(10%)
Scitex - Image	*	unsign 8	unsign 8		
Xerox	*(64 bit)	sign 16			
DVI	*	8, 12, 16, 32	unsign 8, 12 sign		*(20% V3.4)
EPG	*	8 + 16 - sign/unsig		*	

Nearly all customers showed willingness to use vector instructions, definitely if they work upon packed elements in 64 bit registers.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit



Multimedia Focus
Team

X86 64-bit Multimedia ISA Definition Process

mVI

iMultimedia

64-bit Multimedia ISA Ratification Summit

B. Eitan, A. Peleg - 4/92

Intel Confidential

1 of 15

Definition Teams

Definition process done by 2 Bodies:

MMFT - Multimedia Focus Team - Direction steering and ratification

MMEV - Multimedia Evaluation Team - Definition and studies, first cut decisions

Team Participants:

MMFT

P67 IDC - B. Eitan, A. Peleg, M. Kagan, I. Kazachinsky

P67 SC- B. Maytal, M. Mittal, L. Mennemeier

ISWP - K. Gottlieb, SWFG - M. Sundaresan, G. Zalic

960 - K. Smith, R. Brunner, D. Attalah

MMPO - K. Harney, E. Kowashi

EPG - C. Dulong

MMAD - E. Hannah

P6 - A. Glew

MMEV

IDC - A. Peleg, B. Eitan

SC - L. Mennemeier, M. Sundaresan

J. Awakoaiye, M. Mittal.

B. Eitan, A. Peleg - 4/92

Intel Confidential

2 of 15

64-bit Multimedia ISA Ratification Summit

Evaluation Process Objectives & Method

- Ratify the Strawman Multimedia ISA and its applicability to a chosen representative algorithm suite by coding the inner loops of the algorithms in MM ISA and comparing to coding with regular core ISA.
- MM Architecture has large set of open issues. Open issues divided into performance open issues e.g. 8x8 vector multiply, or programming model open issues e.g. rounding mode on multiply.
- Performance open issues are evaluated against an algorithm suite for performance enhancement (hand coding).
- Programming model open issues are evaluated by gathering information about importance from experts (internal & external customers) and performance impact.

B. Eitan, A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

3 of 15



MultiMedia Applications Priority list

First Priority

Medium Priority

Low Priority

64-bit Multimedia ISA Ratification Summit

Intel Confidential

4 of 15

Chosen Algorithms/Inner Loops

Chosen Algorithm	% of Apps	Multimedia Application Domain
• Bitmap text bitblt	15	2D-graphics (Windows 3.x, NT, iGL)
• Line drawing (straight, curved)	?	2D-graphics (Windows 3.x, NT, iGL)
• Antialiasing lines (smoothing)	?	2D-graphics (Windows 3.x, NT, iGL)
• Edge enhancement filter	20	Image Processing.
• DCT/IDCT	7-18-70	JPEG, MPEG1/2 Video, Px64
• Motion estimation(Block Match)	>30	MPEG1/2 Video, Teleconference(Px64)
• Motion compensation(Reconstruct)	13-38	MPEG1/2 Video, Teleconference(Px64)
• Color-space conversion	15	Image Processing, MPEG video, etc.
• Lossless compression (LZ)	?	Networking.
• DTW+ HMM	60-95	Hand Writing/Speech Recognition.
• Gouraud shading	90	3D-graphics.

64-bit Multimedia ISA Ratification Summit

B. Bitan, A. Peleg - 4/92

Intel Confidential

5 of 15



Multimedia Focus
Team

Multimedia Definition Studies

Results

- Performance Study Results - Plan Of Record MM ISA
- Performance Study Results - "Lion Cub"
- Performance Study Results - Enhancement of Open Issues
- Conclusions
- Memory BW Requirements
- Other important Observations
- C++ Encapsulation and Compilerability
- Future plans

64-bit Multimedia ISA Ratification Summit

B. Eitan, A. Peleg - 4/92

Intel Confidential

6 of 15

MM POR* Performance Study Results

Table 1: Enhancements over scalar code

Study	Scalar Coding	MM POR ISA (Avg)
Text Bitblt	1	1
Line Drawing (Straight & Curved)	1	1
Anti-aliasing and Image compositing	1	1.6-6.1
Image Processing - Edge enhancement Filter	1	3.5 (fp)
DCT/IDCT	1	3.1-4.2
Motion Estimation	1	3.9-5.2
Motion Compensation	1	2.8 - 3.9
Color Space Conversions	1	5.8
Lossless compression	1	1
Recognition Algorithms	1	4-5.7
3D - Gouraud shading - interpolation inner loop	1	6

Results are for hand-coding the MM-ISA into hand optimized Gnu P{67 compiler output.

* - 4 issue, 4 Alu, up to 2 mem accesses, one multiply, 2 shifters, 1 merge/pack unit.

B. Eitan, A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

7 of 15

"Lion Cub" Results

- Assuming one issue machine - Calculated from instruction counts.

Study	Scalar Coding	MMPOR ISA (Avg)
Text Bitblt	1	1
Line Drawing (Straight & Curved)	1	1
Anti-aliasing and Image compositing	1	1.6-2.5
Image Processing - Edge enhancement Filter	1	2.5
DCT/IDCT	1	2.5-3.6
Motion Estimation	1	2.7-3.5
Motion Compensation	1	2.5-2.6
Color Space Conversions	1	5.1
Lossless compression	1	1
Recognition Algorithms	1	3.8-4.7
3D - Gouraud shading	1	4

64-bit Multimedia ISA Ratification Summit

B. Eitan, A. Peleg - 4/92

Intel Confidential

8 of 15



Multimedia Focus
Team

Open Issues Performance Enhancement Study Results

Table 1: Performance open issues - Enhancement over MM POR

Study	32-bit data type	8x8 multiply	Vector mul scalar-Xrep	Ximul low order bits	Unsigned right shift	Misalign support	Xmin/Xmax	Xabsolute	Xaverage
Text bitblt						85% (cst)			
Line draw	0%								
Anti-alias		100-140%	70%		0%				0%
Image filter		100%	0%		0%				0%
DCT/IDCT	30%		0%	41%			30%		
Motion est'						15% (load)	23%	14%	
Mot' comp'					5-10%	30-50% (ld)			0%
Color space		210%							
Lossless						0%			
Recognition		15%				10% (load)	6-12%	0%	
3D shading						70% (cst)	5-10%		

The large enhancement available through a 8x8 multiply also results in most of the above cases in a precision loss & quality degradation.

64-bit Multimedia ISA Ratification Summit

B. Eitan, A. Peleg - 4/92

Intel Confidential

9 of 15



- Applications and algorithms for studies were chosen to reflect and be representative of current and future CSC needs.
- MM POR ISA (hand coding) is adequate to enhance most MM algorithms and applications linearly and in some cases super linearly in comparison to scalar coding.
- Out of the applications studied, 2D and 3D operations benefit from misalignment support and Cst. Other than that they do not require the computational enhancement supplied by the MM ISA.
- Performance improving open issues exist. Inclusion in ISA will be decided in summit sessions in the next 2 days.

- 3.5 persons for 2 years (P67 project time).

10 of 15

 SCEA-1423589

CONFIDENTIAL - OUTSIDE COUNSEL EYES ONLY

E 2-04-CV-120
Atto...ys' Eyes Only
F10E5D00047010

Memory Bandwidth Needs

Table 1: Memory BW needs (assuming cache line write packetizing)

Study	32 byte Line Accesses per P67 Cycles (HW depend)	BW limited	% of 32 byte Line Accesses per Ops (HW independent)
Text Bitblt	2/20	Yes	2/4 = 50%
Line Drawing	1/4	Yes	1/8 = 13%
Image composite	4/70	No	4/48 = 8%
Edge enhance	4/120	No	4/144 = 3%
DCT/IDCT	2/242	No	2/216 = 1%
Motion Est'	32/9520	No	32/1584 = 2%
Motion Comp'	3/82	No	3/42 = 7%
Color Space Conv'	6/112	No	6/312 = 2%
Lossless compress	1/300	No	1/96 = 1%
Hw Recognition	6/140	No	4/132 = 5%
Gouraud shading	2/25	Yes	2/30 = 7%

Most MM algorithms have a regular row memory access pattern, thus, one load cache miss generates a line fill used in next 3 iterations. Write packetizing can further decrease memory BW.



Other Important Observations

- MM studies showed register pressure in spite of careful scheduling: Image processing - edge enhancer (45 regs - ~10% - perf), DCT/IDCT (29 regs - Marginal) and Handwriting recognition loops (36 regs - ~20% - perf), Motion Estimation (37 regs - ~40%), Motion Compensation (40 regs - ~25% - perf).
- Register usage will grow with higher parallelism.
- Implementation Issue - The ability to issue 2 Xmerge instructions per cycle is needed in at least 3 of the studies - Anti-aliasing study, recognition operations and DCT/IDCT.

B. Eitan, A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

12 of 15

C++ Encapsulation

- C++ class definitions and operations prototype completed. 2 versions exist with identical interface:
 - 1) Based on 64 bit MM ISA - for use on X86 processors with MM ISA.
 - 2) Emulated vector operations for other processors.
- 2 application inner loops coded in C++ and running (edge enhance filter + Image compositing).
- C++ code compiled via GNU compiler to produce (through ASM inlining) 64-bit assembler code based on MM ISA and running through 64 bit ISA simulator.
- Further work being invested on optimizing the inlined MM ISA.

C++ high level programming -> High performance 64 bit multimedia code is working !!

We believe performance speedup seen in previous foils (hand-coding) can be reached using C++.

64-bit Multimedia ISA Ratification Summit

B. Bitan, A. Peleg - 4/92

Intel Confidential

13 of 15



Vectorizing Compilerability

All C inner loops collected for studies were looked at by GNU compiler group.
Some are vectorizable and some are not.

Gnu Compiler Group opinion

- * All technology to vectorize the integer inner loops does NOT exist today.
- * Most inner loops have to be labeled with "Pragma" to resolve memory disambiguity.
- * Need compiler to highly optimize code before vectorizing can begin (avoid noise).

Bottom line Estimate - In vectorizable cases, only **50%** of performance in comparison to hand coding can be reached by compiler.

64-bit Multimedia ISA Ratification Summit

B. Eitan, A. Peleg - 4/92

Intel Confidential

14 of 15



MM
Multimedia Focus
Team

Future Plans

1) Benchmarks

Currently we have:

- Compression, Decompression - JPEG - MPEG1 Audio (industry standards).
- LZ compression from SPEC (industry standard).
- Kodak Image Processing Benchmarks.
- Hand writing Recognizer.

We will re-code inner loops - Get speed-ups of applications over scalar code.

2) More Customer Visits - make sure we have correct definition.

3) Enlarging the study base - Further cover the MM application domain.

4) SW + Compilers - C++ classes prototype already developed, and compiled to generate high performance MM ISA. Assembler Libraries.

B. Eitan, A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

15 of 15



Session #1 - Multiply Opens

8 x 8-bit Multiply

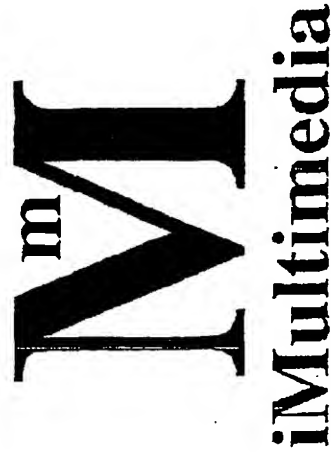
Unsigned Multiply

Results from Low order bits & Rounding Ximul

Vector times Scalar

Xreplicate (also in Session #4)

Chair : L. Mennemeier



64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

1 of 11

8 x 8-bit Multiply

Description: Perform 8 byte-multiplies and produce 8 results from high-order bits.

- Uses:**
- 1) Image Processing Filters (2x if precision is enough, but not likely).
 - 2) Antialiasing Color Lines (2x if precision is enough).
 - 3) Image Compositing (2.4x if precision is enough, not likely).
 - 4) ~15% performance over MM POR ISA in Recognition (loose net correlations).
 - 5) Color Space Conversion (2.6x with loss of 1-bit precision, probably O.K. in many cases: 8- and 16-bit pixels).

Implementation Complexity:

Major microarchitectural change in integer multiply unit. 120K transistors over 100K that implement integer multiplier. 3 cycles, critical timing!

Effort: Large.

64-bit Multimedia ISA Ratification Summit

L. Menneemejer - 5/92

Intel Confidential

2 of 11

8 x 8-bit Multiply (cont.)

Emulation:

Xmerge	src1, zeros, t1	shr	src1, 32, src1
Xmerge	src2, zeros, t2	shr	src1, 32, src1
Ximulw	t1, t2, t3	Xmerge	src1, zeros, t1
Xmerge	src1, zeros, t2		
Ximulw	t1, t2, t4		

5 cycles for 16-bit multiplications (2 more to pack results).

Could be pipelined to 2 cycle throughput with 2 Xmerge units, but then uses up 4 superscalar slots per cycle.

Recommendation: Not to add Ximulb.

64-bit Multimedia ISA Ratification Summit

L. Menne-meier - 5/92

Intel Confidential

3 of 11



Multimedia Focus
Team

Unsigned Multiplies (Xmul)

Description: Produce 4 16-bit (or 8 8-bit) products of unsigned sources.

Uses: 1) Image Compositing.
2) Color Space Conversion.

Implementation Complexity:

Minor microarchitectural change in integer multiply unit. Not critical timing.

Effort: Negligable.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

4 of 11



Multimedia Focus
Team

Unsigned Multiply (cont.)

Emulation:

and src1, msk, t1 and src2, msk, t2 andnot src3, msk, t3 andnot src2, msk, t4
shr t3, 1, t3 shr t4, 1, t4 ximul t1, t2, t5
ximul t1, t4, t6
ximul t2, t3, t7
ximul t3, t4, t8 xshl t6, 1, t6
xshl t7, 1, t7 xadd t6, t5, t6
xshl t8, 2, t8 xadd t7, t6, t7
xadd t8, t7, result

8 cycles on the P67's single multiplier.

Recommendation: To add Xmul for unsigned data.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

5 of 11

Result from Low Order Bits & Rounding Ximul

Description: Produce the low order 16-bit result from a signed integer multiplication.
Do not trap on overflow, and do not saturate.

Uses: 1) IEEE Std. 1180 compliance for Inverse DCT in MPEG Motion Video and Px64 Teleconferencing (3.1x over Scalar Code, and 41% better performance than converting to 64-bit multiplies with MM ISA POR).
2) General support for rounding emulation, which is necessary in some applications. One kind of rounding is not generally satisfactory.

Implementation Complexity:

Add 2-to-1 mux to select result in integer multiply unit. Negligible increase over 100K that implements POR multiplier. Not critical timing.

Effort: Small.

Rounding requires 16-bit incrementers in series to multiplier. Large effort and area budget. Additional cycle added to execution latency.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

6 of 11



Multimedia Focus
Team

Ximul.low (cont.)

Emulation:

and src1, mask, t1 shl src1, 16, t2 shl src1, 32, t3 and src2, mask, t5
shl src2, 16, t6 shl src2, 32, t7 and t2, mask, t2 and t3, mask, t3
and t5, mask, t5 and t6, mask, t6 shl t3, 16, t4 mul t1, t5, t1
shl t6, 16, t8 mul t2, t6, t2
mul t3, t7, t3
mul t4, t8, t4 xpackqw t1, t3, t1
xpackqw t2, t4, t2
xmerge t1, t2, result

12 cycles on the P67, 19 instructions. No MM rounding support for multiplication.

Recommendation: To add Ximul.low.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

7 of 11



Vector Multiplied by Element

Description: Scale all elements of one vector source by just one element from the other source. Put the high order bits from each product in a vector destination register.

Uses: 1) ~69% performance over Antialiasing Color Lines (could also be gained by additional merge units for replicating).
2) This is also the operation used in many other Multimedia applications, but replication does not impact performance.

Implementation Complexity:

Additional state in FSM of integer multiply unit, input muxing. Negligible increase over 100K transistors that implement multiplier. Adding critical timing.
Effort: Medium.

64-bit Multimedia ISA Ratification Summit

L. Menemeyer - 5/92

Intel Confidential

8 of 11

Vector Multiplied by Element (cont.)

Emulation:

(for 16 bit elements)

Xmergew src2, src2, t2

Xmergew t2, t2, t2

Ximulw src1, t2, result

(for 8-bit elements if ximulb provided)

Xmergeb src2, src2, t2

Xmergew t2, t2, t2

Xmergew t2, t2, t2

Ximulb src1, t2, result

Emulation is at least 3 cycle dependency chain. With more Xmerge units a one cycle throughput for can be achieved, thus for repeated, use same throughput of having an Ximulalpha instruction is reached.

For many applications src2 is constant. These can be replicated by compiler.

Recommendation: Not to add Ximulalpha.

64-bit Multimedia ISA Ratification Summit

L. Menemeyer - 5/92

Intel Confidential

9 of 11

Xreplicate (Depends on Vector mul Element)

Description: Replicate low order byte or word into the full 64-bit vector.

Uses:

- 1) Fast matrix multiply done in the column major fashion. Enables throughput of one cycle per multiply accumulate.
- 2) 1.7X performance enhancement over MM POR ISA for anti-aliasing,
- 3) Other algorithms can overcome the need for replicating by utilizing intra pixel parallelism instead (problem for 1 issue "lion cub").
- 4) Compilers can use the instruction for implementing switch statements that are implemented as if-then-else sequences. Also, used in routines that are called many times like short character string compares (20-30 chars) where the cost of emulating replicate is high.

Implementation Complexity:

Enlarge muxing from 4->1 to 5->1 on the Xmerge/Xpack unit. 20% area increase over a 10K transistor Xmerge/Xpack unit. No effect on frequency.

Effort: Small.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

10 of 11

Xreplicate (cont.)

Emulation:

(for 16 bit elements)

Xmergew src, src, t1

Xmergew t1, t1, result

(for 8 bit elements)

Xmergeb src, src, t1

Xmergew t1, t1, t1

Xmergew t1, t1, result

Emulation is at least 2-cycle dependency chain. With more Xmerge units a one cycle throughput can be achieved--for repeated replications, same throughput as having an Xreplicate instruction.

Recommendation: To add Xreplicate.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

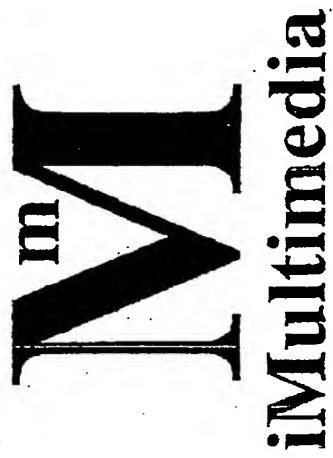
11 of 11



Session #2 - Saturation Features

Saturation Roadmap - X[i]pack vs. Xshl
X[i]pack detailed definition
Saturation Control in opcode
Xmin/Xmax instructions

Chair : A. Peleg



64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

1 of 10



Multimedia Focus
Team

Saturation Roadmap

Three kinds of Saturation:

- Conversion from 16 bit signed/unsigned accumulation to 8 bit unsigned. 16 bits used for higher precision - No saturation in intermediate calculations e.g. Image Processing.
- 8 bit/16 bit interpolations e.g. Intensity interpolation, 3D shadings.
- Special limits e.g. 12 bit signed DCT/IDCT.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

2 of 10

Saturation Roadmap (cont.)

Solutions

Proposed Solution 1:

- $X[i]_{\text{packwb}}$ - Pack low order bits with saturation.
- $X[i]_{\text{addb/w-X}[i]_{\text{subb/w}}}$ - Saturate on overflow/underflow.
- Emulate special case saturation - a) MM POR ISA.
b) $X_{\text{min}}/X_{\text{max}}$.

Need logical shift right to ensure correct saturation on packing.

Proposed Solution 2:

- X_{shlw} - Saturating shift left.
- $X[i]_{\text{addb/w-X}[i]_{\text{subb/w}}}$ - Saturate on overflow/underflow.
- Less special cases. Those left emulated with $X_{\text{min}}/X_{\text{max}}$.

Need to choose between: $X[i]_{\text{pack}}$ or X_{shlw} with saturate.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

3 of 10



Multimedia Focus
Team

Saturation Roadmap (cont.)

X[i]packwb :

Details

- Converts from 16 bit signed (xipack) and 16 bit unsigned (xpack) only to 8 bit unsigned (U->U, S->U).
- Shift down to eliminate fraction part before packing - Use logical shift right.
- Saturation occurs on conversion if integer part overflows lower 8 bits.
- Implementation Complexity - Logical combination of 8 high order bits and sign bit (in signed conversion). Result muxes between src, 0x00 and 0xff. No critical path. Less than 0.5clk.

Xshlw with saturate :

- Possible Saturation cases are: U->U, S->U, S->S. Thus, must specify if src signed/unsigned and destination signed unsigned.
- Shift up to bring integer part to high order bits.
- Saturation occurs on shift if significant bits are shifted out.
- Implementation Complexity - Logical combination of ALL possible high order bits and MSB bit. Result muxes between src, and signed and unsigned limits (5). Unify all xshl possibilities. Critical path in P67 implementation upon shifter - Frequency 0.7clk -> 0.95clk for int and MM shifts. 2.5K tr = 45% increased area

A. Peleg - 4/92

Intel Confidential

4 of 10

64-bit Multimedia ISA Ratification Summit

Saturation Roadmap (cont.)

Conclusions

- X[i]pack much easier to implement.
- In future implementations more probable to have more x[i]pack units than shifters.
- Both solutions do not take care of all special case saturations, although xshlw takes care of all power of 2 cases. Special cases can be emulated by MM ISA (4 instructions, 2 deep) or adding Xmin/Xmax (2 instructions, 2 deep).

Recommendation: Solution 1 + 1.b (adding Xmin/Xmax for emulating special cases and xshr).

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

5 of 10

X[i]packpp Definition

X[i]pack was ratified as a better solution than Xconv (which is in strawman):

- More effective on packing (double throughput).
- Decouples scaling from packing (scaling is not always needed).
- Much easier to implement.
- For unpacking it was shown that xmerge is as effective than Xconv.

Proposed definition:

```
X[i]packwb  src1, src2, dst
dest[63, 56] = saturate(src2[63, 48], 8)
dest[55, 48] = saturate(src2[47, 32], 8)
dest[47, 40] = saturate(src2[31, 16], 8)
dest[39, 32] = saturate(src2[15, 0], 8)
dest[31, 24] = saturate(src1[63, 48], 8)
dest[23, 16] = saturate(src1[47, 32], 8)
dest[15, 8]  = saturate(src1[31, 16], 8)
dest[7, 0]   = saturate(src1[15, 0], 8)
```

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

6 of 10

X[i]packpp Definition (cont.)

Proposed Behavior:

- xpackwb (unsigned version) src1+2 are regarded as unsigned. xipackwb (signed version) src1+2 are regarded as signed.
- Result is always 8 bit unsigned.
- Xpack takes values from LOW order byte of 16 bit element.
- Saturation occurs on conversion to two limits only: 0x00, 0xff.
- Saturation is always active.

Ousted Features:

- No X[i]pack(qw,qb). Packing from scalar does not need saturation and can be done with Xmergeb/w.
- X[i]packwb from high order byte of 16 bit element. In cases where integer part ends up in high order bits (Compositing, 3D shading ??), shift down is necessary before packing. In Compositing this costs 7% degradation in inner loop.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

7 of 10



Multimedia Focus Team

64-bit Multimedia ISA Ratification Summit

- In Strawman saturation is controllable per data type by control bits in control reg.
- Each data type has one enable saturation bit and one sticky status bit.

Proposed change: Add saturation control to opcode space (`x[i]add/x[i]sub` only).

Pro's:

- Saturation is done per operation not per data type.
- Competition does it in opcode. M88110 have saturating add and subtract. DEC Alpha moved overflow control to opcode space.

Con's:

- Addition is one bit to minor opcode field of MM instructions. Space exists as there are no immediates in MM instructions.
- No real need for frequent changes in saturation control was found.

A. Peleg - 4/92

Intel Confidential

8 of 10

Xmin/Xmax Instructions (also in Session #3)

Description: Vector minimum and maximum operation.

Uses: 1) Fast saturation to ANY limits. Part of the full solution for saturation.

Xmax bottom_limit, value, t1

Xmin top_limit, t1, saturated_result.

~30% performance over MM POR ISA in DCT/IDCT.

2) Used for absolute difference operation:

6-12% performance over MM POR ISA in Recognition inner loops.

7-23% performance over MM POR ISA in Motion Estimation.

3) Vector Minimum, Maximum operations (no real case for this was found).

Implementation Complexity:

Adding src1 and src2 to output mux of Xcmp instruction. 250 transistors over 15000 that implement an ALU. No impact on anyway not critical timing.

Effort: Negligible over Xcmp.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

9 of 10

Xmin/Xmax Instructions (cont.)

Emulation: (xmax): Xcmpge src1, src2, mask

and	src2, mask, t1	andnot	src1, mask, t2
or	t1, t2, result		

Can be pipelined to one cycle throughput, but then uses up 3 superscalar slots per cycle.

Recommendation: To add `xminb/w` and `xmaxb/w`.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

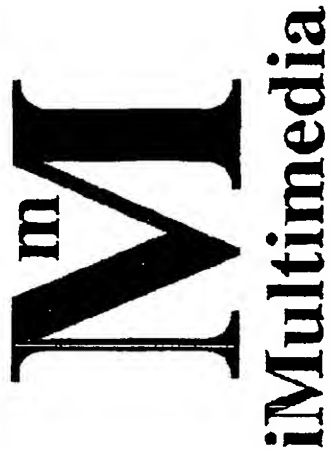
10 of 10



Session #3 - Shift Opens

- Logical Xshr Instruction
- Rounding on Right Shifts
- Immediates in Shifts
- Saturating Xshl Instruction (also in Session #2)
- Extended shift counts

Chair : L. Mennemeier



64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

1 of 11



Multimedia Focus Team

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

2 of 11

Uses:

- 1) Preparation of unsigned values for packing to bytes.
- 2) Divide by unsigned values by powers of two.
- 3) ~10% performance over MM POR ISA in Motion Co

Implementation Complexity:

Adding input to mux at integer shifter's output pad. No effect on shifter area. No impact on timing.

Effort: Negligible over Xsar.



Multimedia Focus
Team

Logical Xshr Instruction (cont.)

Emulation:

Xcmpew t0, t0, mask Xsar src1, count, result xor t0, t0, t0
isub t0, count, count
iadd count, 16, count
Xshl mask, count, mask
andnot src1, mask, result

5 cycles - 4 for creating mask from count, 2 if mask is already available.

If shift count is known, constants can be created for mask. Could be pipelined to a throughput of 1-per-cycle for large data set.

At least for 8-bit data, unsigned is most interesting (not signed).

Recommendation: To add Xshr.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

3 of 11

Rounding on Right Shifts (Xsar)

Description: Shift data elements right by shift count and round to nearest.

- Uses:**
- 1) Divide fixed-point values by powers of two with rounding.
 - 2) Small performance inc. over MM POR ISA in Motion Compensation.

Implementation Complexity:

Adding rounding logic to low order bits, and incrementer to integer shifters' output pads. Large amount of transistor area required over 5K transistors that implement a shifter. Timing critical!

Effort: Substantial over Xsar.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

4 of 11



Rounding on Xsar (cont.)

Emulation:

Xcmpew t1, t1, t1 xor t0, t0, t0
Xsarw src1, cminus1, t2 Xisub t0, t1, t1
Xiadd t1, t2, t2
Xsarw t2, 1, t2

4 cycles - 2 for creating constant, 3 for shift and round if constant is available.
More complex rounding modes require a few more cycles, but would not be supported by this instruction any way.

Recommendation: Not to have rounding on shifters.

64-bit Multimedia ISA Ratification Summit

L. Menne-meier - 5/92

Intel Confidential

5 of 11



Immediates in Shift Instructions

Description: Shift data elements right (or left) by immediate shift count.

Uses: 1) Saves a cycle and a register for misalignment (bit count to byte count).
2) ~7% performance over MM POR ISA in Motion Compensation.

Implementation Complexity:

Muxes for selecting src2 from register or immediate are already present in register file. Additional HW is not needed. No impact on critical timing.

Effort: Negligible.

64-bit Multimedia ISA Ratification Summit

L. Mennemeier - 5/92

Intel Confidential

6 of 11



Multimedia Focus
Team

Immediates in Shift Instructions (cont.)

Emulation:

mov immediate, src2
Xshlp src1, src2, result

Just 2 cycles. Uses at least 1 more register.

Recommendation: Not to add Immediate shift counts.

64-bit Multimedia ISA Ratification Summit

L. Meniemi - 5/92

Intel Confidential

7 of 11



Saturating Xshl Instruction

Description: Shift data elements right by shift count and zero-fill high order bits.

- Uses:**
- 1) Clipping results before packing/storing.
 - 2) 13-20% performance in DCT/IDCT.
 - 3) ~18% performance over MM POR ISA in Color Conversion.

Implementation Complexity:

Add saturation for signed-signed, signed-unsigned, unsigned-unsigned--additional OR/NAND gates at each shifted-out bit at each shifter stage, 5-to-1 muxes to select results, etc. About 2.5K transistors over 5K that implement a shifter. Critical timing (from 0.7 cycles to 0.95 cycles)!

Effort: Large effort--Additional circuit design to meet criticality, extended testing.

64-bit Multimedia ISA Ratification Summit

L. Memmeier - 5/92

Intel Confidential

8 of 11

Saturating Xshl Instruction (cont.)

Emulation: Xcmplew src1, max, t1 Xcmplew src1, min, t2
 and t1, max, t3 and t2, min, t4 or t1, t2, t5
 andnot src1, t5, t5 or t3, t4, t4
 or t4, t5, result

4 cycle dependency chain added to computation (8 instructions).

Could be reduced to 2 cycles with Xmin/Xmax instructions (also only 2 instructions). For example:

Xmin src1, max, t1
 Xmax t1, min, result

Recommendation: Not to add Saturation for Xshl. Should add Xmin, Xmax instead.

L. Mennemeier - 5/92

Intel Confidential

9 of 11

64-bit Multimedia ISA Ratification Summit

Description: Use high order bits of shift count to shift data out when the shift count is greater than the data size.

Uses: 1) For shift counts same as integer side gives the same result as the least significant end of a 64-bit register.

- 2) 5% - 15% performance over MM POR ISA for Motion Compensation.
- 3) 2% - 5% performance for fractional displacements in Motion Estimation.
- 4) Alignment overhead reduced if 1 more bit used in integer shifts.

Implementation Complexity:

If it is decided to use all 64 bits from the src2 register, could require large microarchitectural changes to integer shifter. If only low order 4 bits (for bytes) and 5 bits (for words) are used, the complexity is less.

Effort: Medium to Small depending on definition chosen.

L. Mennemeier - 5/92

Intel Confidential

10 of 11

64-bit Multimedia ISA Ratification Summit



Extended Shift Counts (cont.)

Emulation: (for just full range of register)

cmpg count, 16, mask Xsarw src1, count, t1 Xcmpg zeroes, src1, signs
and mask, signs, t2 andnot t1, mask, t1
or t1, t2, result

3 cycle dependency chain or could be performed with a branch. In Motion Compensation, and Motion Estimation the operations could be performed with the multiplier instead of the shifters.

Recommendation: To provide shift counts for range of register (1 more bit)?

L. Mennemeier - 5/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

11 of 11



Session #4 - New Instructions

Xmin/Xmax (also in session #2)

Xreplicate (also in session #1)

Xaverage

Xdistance/Xabs

Pixel Conversion Support

Chair : A. Peleg

^mMultimedia

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

1 of 10

Xmin/Xmax Instructions (also in Session #2)

Description: Vector minimum and maximum operation.

Uses: 1) Fast saturation to ANY limits. Part of the full solution for saturation.

Xmax bottom_limit, value, t1

Xmin top_limit, t1, saturated_result.

~30% performance over MM POR ISA in DCT/IDCT

2) Used for absolute difference operation:

6-12% performance over MM POR ISA in Recognition inner loops.

7-23% performance over MM POR ISA in Motion Estimation.

3) Vector Minimum, Maximum operations (no real case for this was found).

Implementation Complexity:

Adding src1 and src2 to output mux of Xcmp instruction. 250 transistors over 15000 that implement an ALU. No impact on anyway not critical timing.

Effort: Negligible over Xcmp.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

2 of 10



Xmin/Xmax Instructions (cont.)

Emulation: (xmax): Xcmpge src1, src2, mask
and src2, mask, t1 andnot src1, mask, t2
or t1, t2, result

Can be pipelined to one cycle throughput, but then uses up 3 superscalar slots per cycle.

Recommendation: To add xminb/w and xmaxb/w.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

3 of 10

Xreplicate (Depends on Vector mul Element)

Description: Replicate low order byte or word into the full vector.

- Uses:**
- 1) Imperative for fast matrix multiply done in the column major fashion.
Enables to reach through put of one cycle per multiply accumulate.
 - 2) Gives a 1.7X performance enhancement over MM POR ISA for anti-aliasing,
This performance gap can be eliminated (future implementations) by having more than one xmerge unit.
 - 3) Other algorithms can overcome the need for replicating by utilizing intra pixel parallelism instead (problem for 1 issue "lion cub").
 - 4) Compilers can use the instruction for implementing switch statements that are implemented as if-then-else sequences. Also, used in routines that are called many times like short character string compares (20-30 chars) where the cost of emulating replicate is high.
 - 5) More general and generic than Ximul open issue of vector multiplied by low order element.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

4 of 10

Xreplicate (cont.)

Implementation Complexity:

Belongs upon the Xmerge/Xpack unit. In P67 the addition is to enlarge muxing from 4->1 to 5->1. 20% area increase. No critical timings. Small effort beyond Xmerge/Xpack unit.

Emulation:

(for 16 bit elements)

Xmerge	src, src, t1
Xmerge	t1, t1, result

(for 8 bit elements)

Xmerge	src, src, t1
Xmerge	t1, t1, t1
Xmerge	t1, t1, result

Emulation is at least 2 cycle dependency chain. With more Xmerge units a one cycle throughput put for replication can be achieved, thus, for repeated replication same throughput of having an Xreplicate instruction is reached.

Recommendation: Not to add to architecture. Implementations should provide at least 2 Xmerge/Xpack units.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

5 of 10

Xaverage

Description: Perform the vector operation of $(a+b)/2$

- Uses:**
- 1) Simple $1/2$ pixel interpolations as found in MPEG and Px64.
 - 2) In some cases can be used for UV upsampling - if interpolation is being done and not simple replication.
 - 3) No study showed any performance enhancement by using Xaverage.

Implementation Complexity:

Integer add is used to do $a+b$. Result mux enlarged to be able to do the shift by one.
Easy to implement and no effect on frequency and area compared to xadd.

Emulation: (Assuming logical right shift - Xshr. No overflow but loss of precision)

```
xshr src1, 1, t1    xshr src2, 1, t2
xiadd t1, t2, result
```

Emulation can be pipelined for a one cycle throughput.

Recommendation: Not to add to architecture.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

6 of 10

Xdistance/Xabsolute

Description: Xdistance performs an absolute difference operation in one instruction.
Xabsolute performs only the absolute part.

- Uses:**
- 1) Absolute difference used in MPEG/Px64 block matching motion estimation stage. This stage consumes 26% of V3.4 encoding time.
 - 2) Main time consuming operation of recognition algorithms. 60% - 95% percent of the time is spent on feature comparisons.
 - 3) Values in block matching are 8 bit unsigned and in recognition 8 bit signed/unsigned. Thus, overflow to 9 bits is possible. This means a 16 bit absolute difference is needed.
 - 4) Adding xabsolute can enhance motion estimation by 14% over MM POR, but this is less then the enhancement of Xmin/Xmax (23%).
- In recognition algorithms emulating Xmin/Xmax capability gave the same performance as adding the Xabsolute instruction.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

7 of 10

Xdistance/Xabsolute (cont.)

Implementation Complexity:

- Xdistance is actually 2 instructions in one and thus, cannot be done in one cycle.
- A Xisub has to be done with the result also 2's complemented. Output muxes choose elements from the two results depending on sign of result.
- Effect on area 30% ALU which is 5K tr. Effort is big.
- Xabsolute HW has to look at input sign and perform xiadd src,0 if positive or xisub 0,src if negative.
- No effect on frequency and area. Effort - simple.

Emulation (absolute difference):

xmergeb	src1, zeros, t1	Xmergeb	src2, zeros, t2
xisubw	t1, t2, t3		
xabsw	t3, result		
Xmaxb	src1, src2, t1	Xminb	src1, src2, t2
Xisubb	t1, t2, result		

A. Peleg - 4/92

64-bit Multimedia ISA Ratification Summit

Intel Confidential

8 of 10



Xdistance/Xabsolute (cont.)

Xabs instruction can be emulated by using :

```
Xsar  src, size-1, t1    // extend sign bit to generate mask
Xor   src, t1, t2        // one's complement of negatives
Xisub t2, t1, result     // Two's complement of negatives
```

All solutions can be scheduled via superscalar slots to one cycle throughput.

Recommendation:

In any case not to add xdistance.

In the case of xabsolute:

If xmin/xmax(w/b) are added then there is no need for xabsolute.

If not it is still as fast to emulate xmin/xmax then add an xabsolute instruction.

Thus, recommendation is not to add anything.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

9 of 10

Pixel Format Conversions

Description: ISA support for converting full color (32 bit) pixels to specific formats
e.g. 565 XGA.

Uses: 1) Converting full color pixels to Frame Buffer pixels. In many implementations
(including V3 systems) this is done via a very cheap ASIC.

- Adding instruction to convert to current known formats is too specific.
- General support would be some kind of bit extract or special shift kind of instruction. Emulating the construction with integer POR ISA is a 4 deep dependency chain. With any suggested special support this goes down in best case to 3 cycle chain.

Recommendation: Not to add any special support (Integer team also looked at this
and decided not to add support).

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

10 of 10



Session #5 - Conditional Moves & Misalignment

Conditional Stores

Misalignment - revisited

Chair: M. Sundaresan

**m
VI
iMultimedia**

64-bit Multimedia ISA Ratification Summit

M. Sundaresan - 5/92

Intel Confidential

1 of 6



Multimedia Focus
Team

Conditional Stores

Description:

Selectively update memory with data in the high-order 32 bits of a register, based on the bitmask in the low-order 32 bits of the same register. Each bit in the bitmask could represent bytes, words or double words.

Uses:

- Fast filling of boundary cases for span-line fills.
- Selective updating of pixels in a frame buffer, used for text blits (2D graphics), Z buffering, alpha buffering etc (3D graphics).
- Eliminates a conditional branch per pixel.
- Atleast 4x better than a Read/Modify/Write cycle to partially update memory. Also saves memory bandwidth by not requiring the read cycle.
- Atleast 2-3x better than other software emulations.

M. Sundaresan - 5/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

2 of 6



Multimedia Focus
Team

Emulation:

- Inspect each bit, and update corresponding pixel in a tight loop.
4 clocks per pixel (3x worse than conditional stores).
7 instructions.
- Do a read/modify/write from/to the affected memory, preserving bits that should not be modified, updating those that should be.
1.5 to 6 cycles per pixel (4x worse than conditional stores).
10-13 instruction slots.
- Write multiple pixels in a loop, by switching to an appropriate code fragment depending on the store mask.
3 clocks per pixel (2x worse than conditional stores).
~55 instruction slots.

Recommendation:

To add conditional stores (in byte, word and double word flavors) if compactness and performance for 2D/3D graphics is more important than compilability.

M. Sundaresan - 5/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

3 of 6

Misalignment support (movlo, movhi instructions)**Description:**

These two instructions simulate the first and second bus accesses (along with the correct shifts) which would be done by the hardware if misaligned access is supported by the hardware.

Movlo.[wdq] *reg, addr* loads from the address *addr* after aligning it to the closest double-word, shifts the data to the right appropriately, and puts it in *reg*. The high order bits of the register are zero'd. The store version of the movlo instruction updates only the low order bytes of memory.

Movhi works similarly, except that data accessed is shifted to the left to the correct position, and lower order bits are zero'd. If the memory access is for 0 bytes (eg. when using this instruction with an aligned address), this instruction does not generate any protection violations.

M. Sundaresan - 5/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

4 of 6

Uses:

- Provides a high performance software simulation of misaligned accesses.
- ~50-100% performance improvements for bitblits and other similar 2 operand memory to memory block transfers on the p67.
- Uses only 3 instructions to simulate a misaligned access as opposed to ~10 instructions for a software simulation using shifters.
- Compilers can use these instructions more easily than simulate the software solution.

Emulation:

The following code shows the code to be about 13 instructions, with a dependency chain of 5-7 instructions:

M. Sundaresan - 5/92

Intel Confidential

5 of 6

64-bit Multimedia ISA Ratification Summit



Multimedia Engine

```
long *p, *aligned_p;
long sc_right, sc_left, data_lo, data_hi, data;

aligned_p = (p & ~7); /* address 0 mod 8 */
sc_right = (p & 7); /* right shift count in bytes */

sc_left = 8 - sc_right; /* left shift count in bytes */
sh_right <= 3; /* from bytes to bits */

data_lo = *aligned_p; /* low order bits of data */

if (sh_right) {
    data_hi = *(aligned_p+1); /* high order bits of data */
    sc_left <= 3; /* from bytes to bits */
} else {
    data_hi = 0;
}

data_lo >>= sh_right;
data_hi <= sh_left;

data = data_lo | data_hi;
```

Recommendation:

Strongly recommended on the grounds of performance, compactness, and useability by the compiler.

M. Sundaresan - 5/92

Intel Confidential

6 of 6

64-bit Multimedia ISA Ratification Summit



Session #6 - Other Opens & Desires

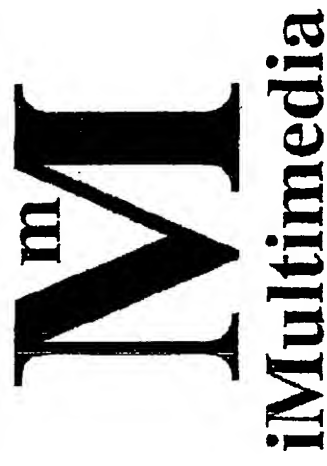
Symmetric compares

Partial Support for 32 bit data type

New Opens from participants

Spills

Chair: M. Mittal



64-bit Multimedia ISA Ratification Summit

M. Mittal - 5/92

Intel Confidential

1 of 4



SCEA-1423643

CONFIDENTIAL - OUTSIDE COUNSEL EYES ONLY

E 2-04-CV-120
Attorneys' Eyes Only
F106800017703

Symmetric Compares

Description: Xcmpccp where $p = b$ (bytes), w (words)

Provide both flavors of comparisons rather than just the list on the left.

<u>cc</u>	<u>relation</u>	<u>cc</u>	<u>relation</u>
e	equal	l	less than signed
ne	not equal	le	less than or equal signed
g	greater than signed	lu	less than unsigned
gu	greater than unsigned	leu	less than or equal unsigned
ge	greater than or equal signed		
geu	greater than or equal unsigned		

Emulation:

xcmpgep src1, src2, dest = xcmplp src2, src1, dest
Same throughput, no performance benefit for symmetry.

64-bit Multimedia ISA Ratification Summit

M. Mittal - 5/92

Intel Confidential

2 of 4



Partial Support for 32-bit Data Type

Description: Provide limited MM ISA support for 32-bit data in the ALUs and/or Merge/Pack Units (support for *add / sub / cmp / pack / merge / shift*).

Uses:

- 1) Audio compression, decompression.
- 2) DCT/IDCT accumulations would be simplified.
- 3) Some customers (e.g. Pixar, Adobe, Mediavision) want to use an intermediate 16.16 packed representation and convert back to an 8.8 packed representation.

Implementation Complexity:

Small

64-bit Multimedia ISA Ratification Summit

M. Mittal - 5/92

Intel Confidential

3 of 4



Multimedia Focus
Team

New Opens from Participants

64-bit Multimedia ISA Ratification Summit

Intel Confidential

M. Mittal - 5/92

4 of 4



Session #7 - Summary and Ratification

Chair : A. Peleg

^mM iMultimedia

64-bit Multimedia ISA Ratification Summit

A. Peleg - 5/92

Intel Confidential

1 of 10



Summary of MM summit

New Additions to POR:

- 1) Unsigned multiply.
- 2) Multiply result from low order bits.
- 3) Closed definition of Vpack as part of the saturation roadmap.
- 4) Saturation control in opcode.
- 5) Logical right shift.
- 6) Extended shift count - Same shift count as integer shift count.
- 7) Recommendation for integer group - if no other misalignment support - enable to shift out all the data in scalar shift instruction.
- 8) Delete of overflow trap on Viadd/Visub.
- 9) Align compares with integer compares - delete the "ne" case.
- 10) Leave opcode space for future expansion to 32 bit data type.

Nearly all additions are marginal beyond the POR and simple to implement.

A. Peleg - 4/92

Intel Confidential

1 of 10

64-bit Multimedia ISA Ratification Summit



3) Saturation control moved to opcode space - only affects $V[i]_{\text{add}}$ and $V[i]_{\text{sub}}$:

- Enough space.
- Frequent changes.
- Competition (and we) do it.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

5 of 10



Session #3 - Shift Summary

Logical Vshr Instruction - Add to ISA.

- Needed for preparation of unsigned values prior to packing.
- Negligible implementation complexity over Vsar.

Rounding on Right Shifts - Don't support HW rounding.

- Very little overhead to emulate.
- Expensive to implement on shifter.

Immediates in Shifts - Don't add immediates.

- Uses up opcode space.
- Emulation overhead is one instruction.

Saturating Vshl Instruction - Don't Add to ISA (closed in Session #2).

Extended shift counts - Use same number of bits as integer side.

- Keeps results the same as scalar code.
- Recommend integer shift enable clearing a register if no other support for misalignment.

A. Peleg - 4/92

Intel Confidential

6 of 10

64-bit Multimedia ISA Ratification Summit



Session #4 - New Instructions

- 1) **Vmin/Vmax - Not To add.**
 - Major performance advantage for absolute difference operation basic to recognition processes.
 - Part of saturation roadmap.
 - Simple to implement.
- 2) **Vreplicate - Not to add.**
 - Too specific.
 - Most benefit can be nullified by supplying 2 Vmerge units.
- 3) **Vaverage - Not to add.**
 - Too specific.
 - Trend towards weighted averaging.
 - Complex to implement.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

7 of 10



Session #4 - New Instructions

- 1) **Vmin/Vmax - Not To add.**
 - Major performance advantage for absolute difference operation basic to recognition processes.
 - Part of saturation roadmap.
 - Simple to implement.
- 2) **Vreplicate - Not to add.**
 - To specific.
 - Most benefit can be nullified by supplying 2 Vmerge units.
- 3) **Vaverage - Not to add.**
 - Too specific.
 - Trend towards weighted averaging.
 - Complex to implement.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

7 of 10



4) **Vdistance** - Not to add.

- Very complex to implement.
- Cannot be done in one cycle
- faster emulated by V_{min}/V_{max} .

5) **Vabsolute** - Not to have.

- Intermediate precision problems on absolute difference.
- Emulated faster by V_{min}/V_{max} .

6) **Pixel Format Conversion Instruction** - Not to have.

- No general support really reduces the dependency chain.
- ASIC can be used - cheap and fast.
- Trend towards true color.

64-bit Multimedia ISA Ratification Summit

A. Peleg - 4/92

Intel Confidential

8 of 10



Session #5 - Memory Interfacing

1) Conditional Store - Not To Add.

- * Saves memory BW (2-3X). This translates to performance mainly in 3D.
- * UGLY.
- * Non-trivial implementation - may affect critical memory access path.

2) Misalignment support - Not to Add - Further evaluation needed.

- * Any support on critical timing of load operation.
- * Performance benefit is available for this fundamental problem with ISA support. Still to be determined if it is a major enhancement.
- * Re-evaluation of the issue looking at new algorithms (Motion Compensation) and an exact memory subsystem.

A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

9 of 10



Session #6 - Other Issues

1) MM Compares -

- * To go with the new change in Integer - delete "ne" case.

2) 32 Bit data type - Not to Add but leave opcode space.

- * Useful for Audio.
- * Pixel interpolation in high quality.
- * Pixar Adobe use it for intermediate calculations
- * Does not assist the 32 bit scalar support.

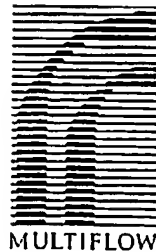
A. Peleg - 4/92

Intel Confidential

64-bit Multimedia ISA Ratification Summit

10 of 10

TRACE /300 Series: F Board Architecture



Multiflow Computer
175 North Main Street
Branford, CT. 06405

December 9, 1988

(203) 488-6090

December 9, 1988

\$Header: janus.mS,v 1.11 88/07/13 13:59:24 lethin Exp \$

Multiflow, TRACE, and Trace Scheduling are trademarks of Multiflow Computer, Inc. UNIX is a trademark of AT&T. The software described in this document is furnished under license and may be used or copied only in accordance with the terms of such license and with the inclusion of the copyright notice shown on this page. Neither the software nor any copies thereof may be provided to or otherwise made available to anyone other than the licensee. Title to and ownership of this software remains with Multiflow Computer, Inc. or with its licensor.

The information in this document is subject to change without notice.

Copyright (C) 1986 by Multiflow Computer, Inc. All rights reserved.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, or transmitted, in any form without prior written permission from Multiflow Computer.

CHAPTER 1

JANUS

1.1 INTRODUCTION

The F board used in /300 series products differs from the /200 F boards in the following important respects:

1. Improved floating point operation latency

- Add/Convert - 3x faster
- Compare - 2.5x faster
- Multiply - 2.5x faster
- Divide - 5.1x faster
- Square Root - 11.4x faster

Reducing the latencies of floating point operations can result in large performance increases for sequential code structures. These reduced latencies arise primarily from using a set of floating point parts from BIT (*Bipolar Integrated Technologies*).

For code with an adequate number of data-independent floating point operations to continuously engage the /200 F unit, /300 systems only allow a modest increase in performance by reducing the amount of time spent "winding up" and "winding down" pipelines in unrolled loops.

By reducing the number of operations that can be in flight at any time, /300 systems relieve demands upon the compiler to locate parallelism in quantities necessary for full performance on the machine. Because the pipelines are shorter, fewer data-independent operations are required.

In addition, the shorter pipelines reduce the number of registers tied up at any time waiting for the results of the floating point pipelines, reducing register allocation constraints.

2. Functional Unit Symmetry

Both falu0 and falu1 include a floating point ALU, a floating point multiplier, an integer ALU, and a "Junkbus" (used to generate logically normalized compare results), and offer a symmetric set of arithmetic and logical opcodes.

This is intended to ameliorate the effects of the register file "commissurotomy" by reducing demands to move operands to the subbank that feeds the needed functional unit. With symmetric opcode sets, any subbank can feed any type of functional unit.

Symmetric functional units can also lead to more consistent application performance. Applications are no longer crippled by heavy concentrations of add, multiply, or divide opcodes.

Divide and square root bandwidth are significantly increased. A 28-wide system can perform 8 simultaneous divides or square roots.

3. **Incompatibility.** The /300 F unit will be unable to execute /200 series object code. Opcodes will differ (forced by coding constraints) and the interpretation of fields in the F instructions will differ. /300 opcodes will also be subject to different resource constraints than the /200 F unit.
4. **Flexible Control Structure.** The new floating point components are expected to evolve in speed over the next year. We're trying to generate a control structure that anticipates this and allows backward compatibility.

This flexible control structure makes all opcodes available in "fast" and "normal" modes. Fast mode reduces all operation latencies (except square root and divide) by 1 beat - 33% for the floating adder, by performing a register file read and arithmetic operation in a single beat.

In addition, providing a "delayed" version of all operations allows operations to begin one beat late, removing the restriction that F opcodes must always begin on an early beat. Sequential operations with odd latencies can be fully chained to use register file bypassing.

The functional units will be able to switch between these modes freely, under control of bits in the instruction word.

5. **Rich opcode set.** A rich set of opcodes is available, subject to encoding restrictions. The /300 opcode set provides Min, Max, Scale, Merge, and Normalize opcodes, a complete set of conversion operations, and rotate and shift instructions.
6. **No 64-bit store restriction.** The ability to perform 64-bit stores from all clusters will be provided. Previously, 64-bit stores could only be performed from clusters 0 and 1.
7. **MAC - Multiply Accumulate.** Three instructions, MACINIT, MACADD and MACSUB allow the board, in some situations where performance might be limited by floating point bandwidth, to perform two double precision flops with one instruction. This improves hand-coded performance for large matrices significantly.

CHAPTER 2 ORGANIZATION

2.1 INSTRUCTION STREAMS AND FUNCTIONAL UNITS

Each F unit can execute two new operations every cycle (two beats). These operations are called `falu0` and `falu1`.

Each operation stream can be routed to one of three functional units or ALUs, depending on the opcode. Floating point multiplies, divides, and square root operations are performed on functional units `falu0m` and `falu1m`. All other floating point operations are performed on `falu0a` and `falu1a`. Multiflow integer opcodes are performed on `falu0i` and `falu1i`. It is possible to have a long latency operation in progress on one functional unit, and issue a short latency operation that completes before the first operation -- operations can pass each other in the pipelines.

Pipelines freeze for memory stalls and for cache misses, and drain when a trap occurs.

The floating point, integer, and logical opcode sets and encoding for `falu0` and `falu1` are identical.

GC operations (`gcops`) are only available on `falu0`.

The return opcode can only be initiated on `falu1`. (The return bit can be combined with any `falu1` opcode.)

2.2 DATA PATHS

To be supplied

2.3 FLOATING REGISTER FILE

Each F unit contains a unique bank of 64 32-bit registers. (Cluster 0's registers are referred to as F bank 0, cluster 1's registers are referred to as F bank 1, etc.) Each bank consists of 2 sub-banks of 32 registers of 32 bits. Registers 0 to 31 form sub-bank 0, and registers 32 to 63 form sub-bank 1.

Double precision and "pair" values are stored in adjacent even/odd registers and are referred to as sources for operands and destinations for results by specifying the even register number. The MSWs of double precision values reside in even registers and the LSWs reside in the following odd registers.

Operands for the `falu0` operation stream are sourced from sub-bank 0; operands for the `falu1` operation stream are sourced from sub-bank 1.

No registers are write-protected, however, some registers are reserved for specific purposes. Registers 31 and 63 by convention contain zero. Register 62 is reserved for trap code. User values stored in these registers may not be maintained across interrupts.

2.4 STORE REGISTER FILE

Each F unit also contains a store register file of 32 32-bit registers, serving as an intermediate holding area between functional units and memory. Values are sourced by the store register file onto the store buses when the integer board executes store opcodes.

The trap code uses store registers 28-31 on each cluster; their value is not guaranteed across interrupts.

The following rules apply:

- Double precision values written to store banks must be stored to an even register. A double word store to register 8 will store its values into registers 8 and 9.
- Double precision values can be saved to memory from any even register. A double word save from register 12 saves registers 12 and 13.

The store register file is implemented with the same components as the register file. Internally, there are 64 registers, but the upper subbank mirrors the lower subbank for 32 effective code-visible registers. This mirroring of subbanks allows the choice of store register to be independent of the store bus it will be driven to.

2.5 INTERNAL FLOATING-POINT REGISTERS

Ten control and status registers are located on the F board, distributed in the following fashion:

```
falu0a - mode register
falu0m - mode register
falula - mode register
falulm - mode register
falu0a - interrupt enable register
falu0m - interrupt enable register
falula - interrupt enable register
falulm - interrupt enable register
falu0a - shift count register
falula - shift count register
```

2.5.1 INTERNAL FLOATING POINT MODE REGISTER

The mode register configures the floating point parts. For proper operation, particular fields must be set as specified below, other fields are user-settable. The operating system performs context switching for this register -- operations for reading and writing it are provided. These mode registers are formatted as follows.

falu0a, falu0m, falu1a, falu1m mode register format

Bit(s)	Description
31-14	Reserved - MFC1
13-9	Reserved - BIT
8	SP - Sticky parity bit
7	BM - Borrow mode
6	R1 - Rounding mode, with R0
5	R0
4	IO - IEEE Overflow mode
3	IU - IEEE Underflow mode
2	IP - Ignore parity
1	FF - Freeze flag
0	ID - IEEE/DEC mode

Reserved Reserved bits should be written with zeroes "to retain compatability with future versions of the floating point chip set". *There's no guarantee that these read as zero.*

SP Sticky Parity. Should always be set to zero.

BM Borrow Mode. Should always be set to zero. *Multiflow does not support the CRY flag.*

R1,R0 Rounding mode. Default is 00->"round to nearest." Other available rounding modes are 01->"round to zero." 10->"round to -infinity." and 11->"round to +infinity." *For fix operations that can require "round to zero" mode regardless of the floating point rounding mode, an explicit "fix and round to zero" opcode is provided.*

IO IEEE Overflow mode. Should always be set to zero.

IU IEEE Underflow mode. Should always be set to zero.

IP Ignore parity. Should always be set to one.

FF Freeze flags on interrupt mode. Should always be set to zero.

ID IEEE/DEC mode. Should always be set to zero to operate on and produce IEEE format numbers.

The mode register is writable in user mode. If a process writes the register inadvertently or incorrectly, it may break its own floating point arithmetic, but this will be isolated from other processes by trap code context switches.

In general, all four mode registers will hold identical values.

2.5.2 BIT INTERRUPT ENABLE REGISTER

The interrupt enable register allows a user to select the condition(s) that can activate an interrupt. An interrupt will occur if one or more of the below conditions is true, at least one of the corresponding enable bits is set, and the master interrupt enable bit is set.

falu0a, falu0m, falu1a, falu1m interrupt register format

Bit(s)	Description
31-14	Reserved, MFC1
13	CRY
12	DIVZ
11	dc
10	DEN
9	NaN
8	dc
7	INX
6	INV
5	UF
4	OV
3	ZR
2	N
1	PE
0	IE

- Reserved Should always be written with zeroes.
- CRY Interrupt on carry bit. Should be set to zero.
- DIVZ Interrupt on divide by zero. User selectable.
- dc Don't care. Set to zero.
- DEN Denormalized number interrupt enable. Should be set to zero. *If the mode register is set correctly, denormalized inputs will be flushed to zero.*
- NaN Not a number interrupt enable. User selectable. *However, if a trap is taken due to the NaN flag, it will not be recorded in the external sticky status register.*
- INX Inexact result trap enable. User selectable.
- INV Invalid operation trap enable. User selectable. *This bit will not mask interrupts caused by MV_CHECK or SELECT operations that see NaNs as their result.*
- UF Underflow trap enable. User selectable.
- OV Overflow trap enable. User selectable. This bit will not mask interrupts caused by MV_CHECK or SELECT operations that see INFINITYs as their result. Setting this bit also enables overflow interrupts from integer ALU operations (falu0i and falu1i) on the F board.
- ZR Zero flag interrupt enable. Should be set to zero.
- N Negative flag interrupt enable. Should be set to zero.
- PE Parity error interrupt enable. Should be set to zero.

IE Master interrupt enable. Should be set to 1.

The interrupt enable bit is writable in user state. Processes writing invalid values into the interrupt enable register may cause their process to receive spurious, non-deterministic interrupts.

The sticky flag register only supports the IEEE required exceptions DIVZ, INV, INX, OV, and UF. If other traps are enabled, interrupts may occur but trap code will be unable to determine their cause. See later section describing floating exception handling for further discussion.

2.5.3 SHIFT COUNT REGISTER

The shift count register (SC) contains a 7 bit two's complement number to indicate the count and direction for a shift or rotate operation. Accessing this register is accomplished by using the SCREGx operations to either write bits (6:0) of the X value or read these same bits (sign extended to 32 bits) from the output port. Only falu0a and falu1a contain this register.

If the shift count register contains a value greater than zero for a shift or rotate operation, a shift/rotate left will occur. If the value is zero, no shift will occur and if the value is less than zero a right shift of -SC will be performed.

2.6 FLOATING STATUS REGISTER

An external sticky 16-bit floating status register tracks exception occurrences on the different functional units, and is used to determine the type of floating exception that caused a floating exception trap. It is also used to control aspects of floating point exception behavior. Opcodes are provided to read and write this register.

Floating Status Register

Bits	Name
15	FSR_FALU1_UNF
14	FSR_FALU1_OVF
13	FSR_FALU1_INX
12	FSR_FALU1_INV
11	FSR_FALU1_DIVZ
10	FSR_FALU1_INX_REM
9	FSR_FALU1_RND_REM
8	FSR_ENB_CMPINV_INT
7	FSR_FALU0_UNF
6	FSR_FALU0_OVF
5	FSR_FALU0_INX
4	FSR_FALU0_INV
3	FSR_FALU0_DIVZ
2	FSR_FALU0_INX_REM
1	FSR_FALU0_RND_REM
0	FSR_ALLINT_MODE

Bits are interpreted as follows:

- FSR_FALU1_UNF, FSR_FALU1_OVF, FSR_FALU1_INX, FSR_FALU1_INV, FSR_FALU1_DIVZ: Specify exceptions that have occurred on falu1a and falu1m. UNF = Underflow Exception, OVF = Overflow Exception, INX = Inexact Result Exception, INV = Invalid Operation Exception, DIVZ = Divide by zero exception.
- FSR_FALU1_INX_REM, FSR_FALU1_RND_REM: Specify exceptions that have occurred on falu1a and falu1m.
- FSR_FALU0_UNF, FSR_FALU0_OVF, FSR_FALU0_INX, FSR_FALU0_INV, FSR_FALU0_DIVZ: Specify exceptions that have occurred on falu0a and falu0m.
- FSR_FALU0_INX_REM, FSR_FALU0_RND_REM: Specify exceptions that have occurred on falu0a and falu0m.
- FSR_ENB_CMPINV_INT: IEEE 754 specifies how floating point compare operations should cause exceptions. This bit supplements the mode registers in falu0a and falu1a to enable invalid operation exceptions from floating point compares. During ALLTRAP mode, if this bit is set, compare operations will generate invalid operation interrupts as specified in the 754 spec.

Additionally, this bit is used to enable junkbus sniffing for infinities during !ALLTRAP mode. Set this bit for DELAYTRAP mode. Clear it for NOTRAP mode.

- FSR_ALLTRAP_MODE: If this bit = 1, all floating point exceptions will be recorded in the floating status register, and all unmasked interrupts will occur, regardless of the destination of the operation. If this bit = 0, the board will operate in delayed trap mode, and status will only be recorded and exceptions will only occur if the destination of the operation requires sanitized data. See section on Floating Exceptions and Traps.

CHAPTER 3
OPERATIONS

3.1 OPERATION ENCODING

The `falv0` and `falv1` operations are formatted as follows:

`boxwid = 0.6i; boxht = 0.4i; box invis "falv0:" Box1: box with .nw at last box.sw "Opcode" "7" box "64" "1" box "Fpop" "1" box "Dest" "6" box "Brdest" "1" box "Src1" "5" box "Src2" "5" box "Fast" "1" box "Gcop" "1" box "Dst_bnk" "3" box "Delay" "1" box invis "falv1:" with .nw at Box1.sw box "Opcode" "7" with .nw at last box.sw box "64" "1" box "Fpop" "1" box "Dest" "6" box "Brdest" "1" box "Src1" "5" box "Src2" "5" box "Fast" "1" box "Return" "1" box "Dst_bnk" "3" box "Delay" "1"`

The different fields specify the following characteristics of the operation to be performed:

- Opcode** is combined with the `64` field and the `Fpop` field to select the particular operation performed. In general, the `Fpop` bit is set to one to direct the opcode to either the floating point ALU or the floating point multiplier, and set to zero to direct the operation to the integer ALU. The `64` bit often differentiates double precision operations from single precision operations; this may vary with the operation. See the opcodes section of this document for more details.
- 64** See above description of `Opcode`.
- Fpop** Set to indicate the operation is to be performed on `falvXa` or `falvXm`. See `Opcode`, above.
- Dest** The register to receive the result of the operation. See `Dst_bnk` for more information.
- Brdest** If `Dst_bnk == Local Register file`, setting this bit will cause the operation to write two registers. If `Dest < 32`, the operation will write registers `Dest` and `Dest + 32`. If `Dest >= 32`, this operation will write registers `Dest` and `Dest - 32`.
- Src1** The register to be read as the "X" or `SRC1` operand to the operation. `falv1` reads register `Src1 + 32`.
- Src2** The register to be read as the "Y" or `SRC2` operand to the operation. `falv1` reads register `Src2 + 32`.
- Fast** If set, the operation is to be performed in "low latency" mode. This option is not available for all opcodes. See later section describing the fast bit.
- Delay** Indicates that the operation is to be initiated one beat later than normal. This option is not available for all opcodes.
- Gcop** Set when operation information from `Src1` and `Dest` are to be taken from the operation and forwarded to control the global controller.
- Return** Instructs the GC to perform a "return from subroutine" operation.

Dst_bnk The register bank to receive the result of the operation:

Dst_bnk:
 000 - branch bank
 001 - local floating bank
 010 - local integer bank
 011 - local store bank
 100 - F Bank 0
 101 - F Bank 1
 110 - F Bank 2
 111 - F Bank 3

When **Dst_bnk** specifies the Branch Bank, **Dest** is interpreted as follows:

Dest Interpretation when Dst_bnk = 000
 000000 - Bit Bucket
 000001 - Local I branch bank, (ialu0), element 1.
 through
 000111 - Local I branch bank, (ialu0), element 7.
 100000 - Local falu0i branch bank, element 1
 100001 - Local falu1i branch bank, element 1

Values written to the bit bucket are discarded, no write resources are used. Note, however, that core resources and output resources associated with the opcode will be used.

The local F Bank can be referred to either implicitly, via code 001, or explicitly via its bank number. The former will result in a lower latency to write. The latter will result in a "bus-bounce", using FL bus resources and requiring a single additional beat of latency to write.

Note the differences in the encoding between the /300 and /200 F units: the MV and MNOP bit have been discarded, and the Fast and Delay bits have been added. The ROUT bit has been renamed Fpop and performs a slightly different function than before. (MNOP instructions are provided solely by an opcode).

When branch banks are specified as the destination, they are always written with the LSB of the result. (Note that this differs from the I board, which always writes a logically normalized result to its local branch bank)

3.2 OPCODES

The **falu0** and **falu1** operation streams each control three functional units: a floating point ALU, a floating point multiplier/divider/rooter, and an integer ALU.

The **opcode**, **64**, **Fpop** fields in the operation word select the functional unit and the operation to be performed. These operations are listed below in groups having similar constraints and resource characteristics.

When indicated below, operations can be issued in a delayed manner, by setting the **delay** bit in the operation word. A delayed operation reads its operands and writes its results one beat later than a non-delayed operation, to allow tighter compaction of operations. Operation latency, measured as time from operands to result, is unchanged. For example, higher sequential code throughput is achieved by issuing a chain of data-independent multiplies in the sequence:

```

instr c10 falu0e mpy.64      r0,r2,r4;
instr c10 falu0l mpy.64    r0,r0,r6;
instr;
instr c10 falu0e mpy.64      r0,r0,r8;
instr c10 falu0l mpy.64    r0,r0,r10;
instr;
.
.
.

```

Rather than every other beat, as the same code sequence is most efficiently executed without the delay bit:

```

instr c10 falu0e mpy.64      r0,r2,r4;
instr;
instr c10 falu0e mpy.64      r0,r0,r6;
instr;
instr c10 falu0e mpy.64      r0,r0,r8;
instr;
instr c10 falu0e mpy.64      r0,r0,r10;
instr;
.
.
.

```

It is possible to reduce the latencies of many operations by one beat, by setting the Fast bit in the operation. Operations for falu0i and falu1i cannot have the "fast" bit set; they're already performed in fast, low latency mode.

In the following descriptions, X is the operand selected by src1, Y is the operand selected by src2. Z is the result to be written to dest. For pair operations, X1 is the even register pointed to by src1, X2 is the odd register pointed to by src1|1. Y1 is the even register pointed to by src2, and Y2 is the odd register pointed to by src2|1. Z1 and Z2 are the result, written to dest and dest|1.

In the sections below, the 8 bit opcodes is given in hexadecimal are for the opcode field concatenated to the 64 bit in the operation. Opcodes given in lower case are supported by the TRACE assembler (*as*); opcodes given in upper case are not available via the assembler.

For each group of operations, we give also give a "resource class." The resource describes the TRACE resources used by executing any operation in the class. A table of resource usage classes is given later in this manual.

3.2.1 FLOATING POINT OPERATIONS

These floating point operations are performed on the BIT multiplier and ALU. All opcodes in this section also have the Fpop bit set.

3.2.1.1 Single Precision Multiply Operations

Options	Resource Class	Notes
(None)	RUC_SP_MUL	
DELAY	RUC_SP_DLY_MUL	
FAST	RUC_SP_FST_MUL	
DELAY,FAST	RUC_SP_DLY_FST_MUL	

- mpy.f32 0xa0
 $Z = X * Y$
- mpym.s.f32 0xa2
 $Z = \text{abs}(X) * Y$
- mpymm.f32 0xa4
 $Z = \text{abs}(X) * \text{abs}(Y)$

3.2.1.2 Double Precision Multiply Operations

Options	Resource Class	Notes
(None)	RUC_DP_MUL	
DELAY	RUC_DP_DLY_MUL	
FAST	RUC_DP_FST_MUL	*** NOT AVAILABLE ***
DELAY,FAST	RUC_DP_DLY_FST_MUL	*** NOT AVAILABLE ***

- mpy.f64 0xa1
 $Z = X * Y$
- mpym.s.f64 0xa3
 $Z = \text{abs}(X) * Y$
- mpymm.f64 0xa5
 $Z = \text{abs}(X) * \text{abs}(Y)$

3.2.1.3 Single Precision Division Operation

Options	Resource Class	Notes
(None)	RUC_SP_DIV_MUL	
DELAY	RUC_SP_DIV_DLY_MUL	

FAST divide operations are not available

NOTE

There is a significant difference here from the equivalent /200 operation. On the /300 F board, src1 is the dividend (X) and src2 is the divisor (Y). On the /200, the operands are the other way around.

- div.f32 0xc0
 $Z = X / Y$

3.2.1.4 Double Precision Division Operation

Options	Resource Class	Notes
(None)	RUC_DP_DIV_MUL	
DELAY	RUC_DP_DIV_DLY_MUL	

FAST divide operations are not available

NOTE

There is a significant difference here from the equivalent /200 operation. On the /300 F board, src1 is the dividend (X) and src2 is the divisor (Y). On the /200, the operands are the other way around).

- div.f64 0xc1
 $Z = X / Y$

3.2.1.5 Single Precision Square Root Operation

Options	Resource Class	Notes
(None)	RUC_SP_SQRT_MUL	
DELAY	RUC_SP_SQRT_DLY_MUL	

FAST square root operations are not available

- sqrt.f32 0xc2
 $Z = \text{sqrt}(X)$

3.2.1.6 Double Precision Square Root Operation

Options	Resource Class	Notes
(None)	RUC_DP_SQRT_MUL	
DELAY	RUC_DP_SQRT_DLY_MUL	

FAST square root operations are not available

- sqrt.f64 0xc3
 $Z = \text{sqrt}(X)$

3.2.1.7 Pair Floating Point Multiply Operations

Options	Resource Class	Notes
(None)	RUC_PM_MUL	
DELAY	RUC_PM_DLY_MUL	
FAST	RUC_PM_FST_MUL	*** NOT AVAILABLE ***
DELAY, FAST	RUC_PM_DLY_FST_MUL	*** NOT AVAILABLE ***

- **pmpy.f32 0xb0**
 $Z1 = X1 * Y1, Z2 = X2 * Y2$
- **pmpyms.f32 0xb1**
 $Z1 = |X1| * Y1, Z2 = |X2| * Y2$
- **pmpymm.f32 0xb2**
 $Z1 = |X1| * |Y1|, Z2 = |X2| * |Y2|$

3.2.1.8 Single Precision faluXa operations

Options	Resource Class	Notes
(None)	RUC_SP_ALU	
DELAY	RUC_SP_DLY_ALU	
FAST	RUC_SP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_ALU	Requires fast ALU

- **add.f32 0x08**
 $Z = X + Y$
- **addmm.f32 0x0c**
 $Z = |X| + |Y|$
- **sub.f32 0x0a**
 $Z = X - Y$
- **submm.f32 0x0e**
 $Z = |X| - |Y|$
- **min.f32 0x02**
 $Z = \min(X, Y)$
- **max.f32 0x04**
 $Z = \max(X, Y)$
- **abs.f32 0x06**
 $Z = |X|$
- **scale.f32 0x20**
 $Z = \text{exponent } X + Y$
The integer Y is added to the exponent of X. The sign and mantissa of X are passed unmodified. If this instruction overflows or underflows, the result is always a “wrapped” floating point number. If X is a normalized zero, the result is non-zero. The least significant 8 bits of SRC2 are interpreted as a two’s complement number. Other bits are ignored.
- **merge.f32 0x22**
 $Z = \text{SIGN } X \mid \text{EXPONENT } Y \mid \text{MANTISSA } X$
The integer Y is concatenated with the sign and mantissa field of X.

3.2.1.9 Double Precision ALU operations

Options	Resource Class	Notes
(None)	RUC_DP_ALU	
DELAY	RUC_DP_DLY_ALU	
FAST	RUC_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_DP_DLY_FST_ALU	Requires fast ALU

- **add.f64 0x09**
 $Z = X + Y$
- **addmm.f64 0x0d**
 $Z = |X| + |Y|$
- **sub.f64 0x0b**
 $Z = X - Y$
- **submm.f64 0x0f**
 $Z = |X| - |Y|$
- **min.f64 0x03**
 $Z = \min(X, Y)$
- **max.f64 0x05**
 $Z = \max(X, Y)$
- **abs.f64 0x07**
 $Z = |X|$
- **scale.f64 0x21**
 $Z = \text{exponent } X + Y$
The integer Y is added to the exponent of X. The sign and mantissa of X are passed unmodified. If this instruction overflows or underflows, the result is always a “wrapped” floating point number. If X is a normalized zero, the result is non-zero. src2 is a 64-bit integer (Ie, the even register of a 64-bit register pair. Put Y in the odd register) The least significant 11 bits of SRC2 are tested as a two’s complement signed number. Other bits are ignored.
- **merge.f64 0x23**
 $Z = \text{SIGN } X \mid \text{EXPONENT } Y \mid \text{MANTISSA } X$
The integer Y is concatenated with the sign and mantissa field of X.

3.2.1.10 Conversion operations that take a 32-bit operand and produce a 32-bit result.

Options	Resource Class	Notes
(None)	RUC_SP_ALU	
DELAY	RUC_SP_DLY_ALU	
FAST	RUC_SP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_ALU	Requires fast ALU

- **f32u32**
0x10 SP -> 32-bit unsigned integer
- **f32s32 0x12**
SP -> 32-bit signed integer

- **u32f32 0x18**
32-bit unsigned integer -> SP
- **s32f32 0x1a**
32-bit signed integer -> SP
- **f32u32_rn 0x14**
SP -> 32-bit unsigned integer (Rnd to 0)
- **f32s32_rn 0x16**
SP -> 32-bit signed integer (Rnd to 0)
- **trunc.f32 0x1c**
SP -> Single precision format integer
Both operand and result are SP format floating point numbers. Example, 1.75 -> 2.0
- **trunc_rn.f64 0x1e**
SP -> Single precision format integer (Rnd to 0).
Both operand and result are SP format floating point numbers. Example, 1.75 -> 1.0

3.2.1.11 Conversion operations that take a 32-bit operand and produce a 64-bit result.

Options	Resource Class	Notes
(None)	RUC_SP_DP_ALU	
DELAY	RUC_SP_DP_DLY_ALU	
FAST	RUC_SP_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DP_DLY_FST_ALU	Requires fast ALU

- **u32f64 0x68**
32-bit unsigned integer -> DP
- **s32f64 0x6a**
32-bit signed integer -> DP
- **f32u64 0x60**
SP -> 64-bit unsigned integer
- **f32s64 0x62**
SP -> 64-bit signed integer
- **f32u64_rn 0x64**
SP -> 64-bit unsigned integer (Rnd to 0)
- **f32s64_rn 0x66**
SP -> 64-bit signed integer (Rnd to 0)
- **f32f64 0x6c**
SP -> DP

3.2.1.12 Conversion operations that take a 64-bit operand and produce a 32-bit result.

Options	Resource Class	Notes
(None)	RUC_DP_SP_ALU	
DELAY	RUC_DP_SP_DLY_ALU	
FAST	RUC_DP_SP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_DP_SP_DLY_FST_ALU	Requires fast ALU

- f64u32 0x61
DP -> 32-bit unsigned integer
- f64s32 0x63
DP -> 32-bit signed integer
- u64f32 0x69
64-bit unsigned integer -> SP
- s64f32 0x6b
64-bit signed integer -> SP
- f64u32_rn 0x65
DP -> 32-bit unsigned integer (Rnd to 0)
- f64s64_rn 0x67
DP -> 32-bit signed integer (Rnd to 0)
- f64f32 0x6d
DP -> SP

3.2.1.13 Conversion operations that take a 64-bit operand and produce a 64-bit result

Options	Resource Class	Notes
(None)	RUC_DP_ALU	
DELAY	RUC_DP_DLY_ALU	
FAST	RUC_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_DP_DLY_FST_ALU	Requires fast ALU

- f64u64 0x11
DP -> 64-bit unsigned integer
- f64s64 0x13
DP -> 64-bit signed integer
- u64f64 0x19
64-bit unsigned integer -> DP
- s64f64 0x1b
64-bit signed integer -> DP
- f64u64_rn 0x15
DP -> 64-bit unsigned integer (Rnd to 0)
- f64s64_rn 0x17
DP -> 64-bit signed integer (Rnd to 0)

- `trunc.f64 0x1d`
DP -> DP format integer.
Result is Double Precision, with an integral value.
- `trunc_rn.f64 0x1f`
DP -> DP format integer (Rnd to 0)
Result is Double Precision, with an integral value.

3.2.1.14 Single Precision Compare Operations

Options	Resource Class	Notes
(None)	RUC_SP_CMP	
DELAY	RUC_SP_DLY_CMP	
FAST	RUC_SP_FST_CMP	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_CMP	Requires fast ALU

The destination is written with the 32-bit value "00000001" if the specified condition is true, or zero if false.

- `ceq.f32 0xe0`
Compare (X == Y)
- `clt.f32 0xe2`
Compare (X < Y)
- `cle.f32 0xe4`
Compare (X <= Y)
- `cuo.f32 0xe6`
Compare (X .UO. Y)
Return true if either operand is a NaN.
- `cne.f32 0xe8`
Compare not(X == Y)
- `cnlt.f32 0xea`
Compare not(X < Y)
- `cnle.f32 0xec`
Compare not(X <= Y)
- `cnuo.f32 0xee`
Compare not(X .UO. Y)
Return true if neither operand is a NaN.
- `emeq.f32 0xf0`
Compare (|X| == |Y|)
- `emlt.f32 0xf2`
Compare (|X| < |Y|)
- `emle.f32 0xf4`
Compare (|X| <= |Y|)
- `emne.f32 0xf8`
Compare not(|X| == |Y|)

- **cmnlt.f32 0xfa**
Compare not($|X| < |Y|$)
- **cmnle.f32 0xfc**
Compare not($|X| \leq |Y|$)

3.2.1.15 Double Precision Compare Operations

Options	Resource Class	Notes
(None)	RUC_DP_CMP	
DELAY	RUC_DP_DLY_CMP	
FAST	RUC_DP_FST_CMP	Requires fast ALU
DELAY,FAST	RUC_DP_DLY_FST_CMP	Requires fast ALU

The destination is written with the 32-bit value "00000001" if the specified condition is true, or zero if false.

- **ceq.f64 0xe1**
Double Precision Compare ($X == Y$)
- **elt.f64 0xe3**
Double Precision Compare ($X < Y$)
- **ele.f64 0xe5**
Double Precision Compare ($X \leq Y$)
- **cun.f64 0xe7**
Double Precision Compare ($X .UO. Y$)
Return true if either operand is a NaN.
- **cne.f64 0xe9**
Double Precision Compare not($X == Y$)
- **cnlt.f64 0xeb**
Double Precision Compare not($X < Y$)
- **cnle.f64 0xed**
Double Precision Compare not($X \leq Y$)
- **cnuo.f64 0xef**
Double Precision Compare not($X .UO. Y$)
Return true if neither operand is a NaN.
- **cmeq.f64 0xf1**
Double Precision Compare ($|X| == |Y|$)
- **cmilt.f64 0xf3**
Double Precision Compare ($|X| < |Y|$)
- **cmle.f64 0xf5**
Double Precision Compare ($|X| \leq |Y|$)
- **cmne.f64 0xf9**
Double Precision Compare not($|X| == |Y|$)
- **cmnlt.f64 0xfb**
Double Precision Compare not($|X| < |Y|$)

- **cmnle.f64 0xfd**
Double Precision Compare not($|X| \leq |Y|$)

3.2.1.16 Pair Floating Point ALU operations

Options	Resource Class	Notes
(None)	RUC_PM_ALU	
DELAY	RUC_PM_DLY_ALU	
FAST	RUC_PM_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_PM_DLY_FST_ALU	Requires fast ALU

- **pmin.f32 0x80**
 $Z1 = \min(X1, Y1), Z2 = \min(X2, Y2)$
- **pmax.f32 0x81**
 $Z1 = \max(X1, Y1), Z2 = \max(X2, Y2)$
- **pabs.f32 0x82**
 $Z1 = \text{abs}(X1), Z2 = \text{abs}(X2)$
- **padd.f32 0x83**
 $Z1 = X1 + Y1, Z2 = X2 + Y2$
- **psub.f32 0x84**
 $Z1 = X1 - Y1, Z2 = X2 - Y2$
- **paddmm.f32 0x85**
 $Z1 = |X1| + |Y1|, Z2 = |X2| + |Y2|$
- **psubmm.f32 0x86**
 $Z1 = |X1| - |Y1|, Z2 = |X2| - |Y2|$
- **pscale.f32 0x94**
The integer Y1 is added to the exponent of X1. The sign and mantissa of X1 are passed unmodified to Z1. Z2 is formed similarly from Y2 and X2.
- **pmerge.f32 0x95**
The integer Y1 is concatenated with the sign and mantissa field of X1 to produce Z1. Z2 is similarly formed from Y2 and X2.
- **pf32u32 0x92**
Pair SP -> 32-bit unsigned integer
- **pf32s32 0x93**
Pair SP -> 32-bit signed integer
- **pu32f32 0x89**
Pair 32-bit unsigned integer -> SP
- **ps32f32 0x88**
Pair 32-bit signed integer -> SP
- **pf32u32_rn 0x90**
Pair SP -> 32-bit unsigned integer (Rnd to 0)
- **pf32u32_rn 0x91**
Pair SP -> 32-bit signed integer (Rnd to 0)

- **ptrunc_rn.f32 0x8a**
Pair SP -> SP format integer
- **ptrunc_rn.f32 0x8b**
Pair SP -> SP format integer (Rnd to 0)

3.2.1.17 Pair Floating Point Compare Operations

Options	Resource Class	Notes
(None)	RUC_SP_CMP	
DELAY	RUC_SP_DLY_CMP	
FAST	RUC_SP_FST_CMP	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_CMP	Requires fast ALU

The destinations are written with the 32-bit values "00000001" if the specified conditions are true, or zero if false. The branch bank is not a legal destination for this operation.

- **pceq.f32 0xd0**
Pair Compare (X1 == Y1), (X2 == Y2)
- **pelt.f32 0xd2**
Pair Compare (X1 < Y1), (X2 < Y2)
- **pcle.f32 0xd4**
Pair Compare (X1 <= Y1), (X2 <= Y2)
- **pcuo.f32 0xd6**
Pair Compare (X1 .UO. Y1), (X2 .UO. Y2)
- **pneq.f32 0xd8**
Pair Compare not(X1 == Y1), not(X2 == Y2)
- **penlt.f32 0xda**
Pair Compare not(X1 < Y1), not(X2 < Y2)
- **penle.f32 0xdc**
Pair Compare not(X1 <= Y1), not(X2 <= Y2)
- **penuo.f32 0xde**
Pair Compare not(X1 .UO. Y1), not(X2 .UO. Y2)
- **pcmeq.f32 0xd1**
Pair Compare (|X1| == |Y1|), (|X2| == |Y2|)
- **pcmlt.f32 0xd3**
Pair Compare (|X1| < |Y1|), (|X2| < |Y2|)
- **pcmle.f32 0xd5**
Pair Compare (|X1| <= |Y1|), (|X2| <= |Y2|)
- **pcmne.f32 0xd9**
Pair Compare not(|X1| == |Y1|), not(|X2| == |Y2|)
- **pcmnlte.f32 0xdb**
Pair Compare not(|X1| < |Y1|), not(|X2| < |Y2|)
- **pcmnle.f32 0xdd**
Pair Compare not(|X1| <= |Y1|), not(|X2| <= |Y2|)

3.2.2 BIT INTEGER OPERATIONS

These integer operations are performed on `faluxa`. All opcodes in this section also have the `Fpop` bit set.

3.2.2.1 32-bit Integer Operations

Resource usage is the same as for single precision `faluxa` operations.

Options	Resource Class	Notes
(None)	RUC_SP_ALU	
DELAY	RUC_SP_DLY_ALU	
FAST	RUC_SP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_ALU	Requires fast ALU

3.2.2.1.1 Arithmetic.

- **IADD 0x30**
 $Z = X + Y$
- **ISUB 0x32**
 $Z = X - Y$
- **IADDP 0x34**
 $Z = X + Y + 1$
- **ISUBM 0x36**
 $Z = X - Y - 1$
- **IABSX 0x38**
 $Z = \text{abs}(X)$
- **ISMAX 0x3a**
 $Z = \text{Signed integer max}(X, Y)$
- **ISMIN 0x3c**
 $Z = \text{Signed integer min}(X, Y)$
- **IUMAX 0x3e**
 $Z = \text{Unsigned integer max}(X, Y)$
- **IUMIN 0x40**
 $Z = \text{Unsigned integer min}(X, Y)$

3.2.2.1.2 Bitwise Logical.

- **INAND 0x42**
 $Z = \text{NOT}(XY)$
- **IORNX 0x44**
 $Z = \text{NOT}(X) \text{ or } Y$
- **IOR 0x46**
 $Z = X \text{ or } Y$

- **IAND 0x48**
Z = X and Y
- **INOR 0x4a**
Z = NOT(X or Y)
- **IANDNX 0x4c**
Z = NOT(X) and Y
- **IXNOR 0x4e**
Z = X xnor Y
- **IXOR 0x50**
Z = X xor Y
- **INOTX 0x52**
Z = not(X)

3.2.2.1.3 Shift and Rotate operations. The shift count register (SC) inside *fnuXa* determines the amount to shift. If SC > 0, then shift left by SC; if SC < 0, then shift right by -SC. If SC == 0, no shift is performed.

- **LSS 0x54**
Z = Logical shift X with sticky bit.
Sticky bit: for right shifts, all bits of the result shifted out are ored with the least significant bit position of the result. Zeroes are shifted in during right and left shifts.
- **LS 0x56**
Z = Logical shift X
- **AS 0x58**
Z = Arithmetic shift X.
- **ROTX 0x5a**
Z = Rotate X.
- **ROTC 0x5c**
Z Rotate Y|X (Concatenated).
The 32 bit integers Y and X are concatenated and rotated by the signed two's complement number in the shift count register. Before the rotate, Y is in the most significant word position. The least significant 32 bits of the result is returned.
- **BITR 0x26**
Z = Rotate bit reversed X|X.
The 32 bit integer X is bit-reversed and concatenated with a non bit-reversed X. Before the rotate portion of the operation, the non-bit-reversed portion of X is in the least significant word position. After rotation the least significant 32 bits are returned.

3.2.2.2 64-bit Integer Operations

Resource use is the same as for double precision ALU operations.

Options	Resource Class	Notes
(None)	RUC_DP_ALU	
DELAY	RUC_DP_DLY_ALU	
FAST	RUC_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_DP_DLY_FST_ALU	Requires fast ALU

3.2.2.2.1 Arithmetic.

- **LIADD 0x31**
 $Z = X + Y$
- **LISUB 0x33**
 $Z = X - Y$
- **LIADDP 0x35**
 $Z = X + Y + 1$
- **LISUBM 0x37**
 $Z = X - Y - 1$
- **LIABSX 0x39**
 $Z = \text{abs}(X)$
- **LISMAX 0x3b**
 $Z = \text{Signed integer max}(X, Y)$
- **LISMIN 0x3d**
 $Z = \text{Signed integer min}(X, Y)$
- **LIUMAX 0x3f**
 $Z = \text{Unsigned integer max}(X, Y)$
- **LIUMIN 0x41**
 $Z = \text{Unsigned integer min}(X, Y)$

3.2.2.2.2 Bitwise Logical.

- **LINAND 0x43**
 $Z = \text{not}(XY)$
- **LIONX 0x45**
 $Z = \text{not}(X) \text{ or } Y$
- **LIOR 0x47**
 $Z = X \text{ or } Y$
- **LIAND 0x49**
 $Z = X \text{ and } Y$
- **LINOR 0x4b**
 $Z = \text{not}(X \text{ or } Y)$
- **LIANDNX 0x4d**
 $Z = \text{not}(X) \text{ and } Y$
- **LIXNOR 0x4f**
 $Z = X \text{ xnor } Y$
- **LIXOR 0x51**
 $Z = X \text{ xor } Y$
- **LINOTX 0x53**
 $Z = \text{not}(X)$

3.2.2.2.3 Shift and Rotate operations. The shift count register (SC) inside `aluXa` determines the amount to shift. If `SC > 0`, then shift left by `SC`; if `SC < 0`, then shift right by `-SC`. If `SC == 0`, no shift is performed.

- **LLSS 0x55**
Z = Logical shift X with sticky bit. Sticky bit: for right shifts, all bits of the result shifted out are ored with the least significant bit position of the result. Zeroes are shifted in during right and left shifts.
- **LLS 0x57**
Z = Logical shift X
- **LAS 0x59**
Z = Arithmetic shift X
- **LROTX 0x5b**
Z = Rotate X

3.2.2.3 32 bits in, 64 bits out Integer ALU Operations

Options	Resource Class	Notes
(None)	RUC_SP_DP_ALU	
DELAY	RUC_SP_DP_DLY_ALU	
FAST	RUC_SP_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DP_DLY_FST_ALU	Requires fast ALU

- **LROTC 0x5c**
Z = Rotate Y|X (Concatenated)
The 32 bit integers Y and X are concatenated and rotated by the signed two's complement number in the shift count register. Before the rotate, Y is in the most significant word position. The full 64-bit result is returned.

3.2.2.4 Integer Multiply Operations

Multiply two 32-bit integers, return a 64-bit result.

Options	Resource Class	Notes
(None)	RUC_SP_DP_MUL	
DELAY	RUC_SP_DP_DLY_MUL	
FAST	RUC_SP_DP_FST_MUL	Requires fast MUL
DELAY, FAST	RUC_SP_DP_DLY_FST_MUL	Requires fast MUL

- **IMULT 0xb8**
Z = Unsigned X * Unsigned Y
- **IMULTSX 0xba**
Z = Signed X * Unsigned Y
- **IMULTS 0xbc**
Z = Signed X * Signed Y

3.2.2.5 Integer Multiply Operations, Halfword

Halfword operations multiply two 32-bit integers, and return the least significant 32 bits of the result

Options	Resource Class	Notes
(None)	RUC_SP_MUL	
DELAY	RUC_SP_DLY_MUL	
FAST	RUC_SP_FST_MUL	Requires fast MUL
DELAY, FAST	RUC_SP_DLY_FST_MUL	Requires fast MUL

- **IMULTH 0xa8**
Z = Unsigned X * Unsigned Y, Halfword
- **IMULTHSX 0xaa**
Z = Signed X * Unsigned Y, Halfword
- **IMULTHS 0xac**
Z = Signed X * Signed Y, Halfword

3.2.2.6 Pair Integer ALU operations

Options	Resource Class	Notes
(None)	RUC_PM_ALU	
DELAY	RUC_PM_DLY_ALU	
FAST	RUC_PM_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_PM_DLY_FST_ALU	Requires fast ALU

3.2.2.6.1 Pair Integer Arithmetic.

- **PIADD 0x97**
Pair Z1 = X1 + Y1, Z2 = X2 + Y2
- **PISUB 0x98**
Pair Z1 = X1 - Y1, Z2 = X2 - Y2
- **PIADDP 0x99**
Pair Z1 = X1 + Y1 + 1, Z2 = X2 + Y2 + 1
- **PISUBM 0x9a**
Pair Z1 = X1 - Y1 - 1, Z2 = X2 - Y2 - 1
- **PIABSX 0x9b**
Pair Z1 = abs(X1), Z2 = abs(X2)
- **PISMAX 0x9c**
Pair Z1 = signed integer max(X1, Y1), Z2 = signed integer max(X2, Y2)
- **PISMIN 0x9d**
Pair Z1 = signed integer min(X1, Y1), Z2 = signed integer min(X2, Y2)
- **PIUMAX 0x9e**
Pair Z1 = unsigned integer max(X1, Y1), Z2 = unsigned integer max(X2, Y2)

- **PIUMIN 0x9f**
Pair Z1 = unsigned integer min(X1,Y1), Z2 = unsigned integer min(X2,Y2)

3.2.2.7 BIT Control Register Manipulation

Operations to write and read internal control registers on the BIT parts have the following resource usage:

Operations to write **aluXm** internal state:

Options	Resource Class	Notes
(None)	RUC_WR_MUL	Dest/dst_bnk should point at bit bucket

Operations to write **aluXa** internal state:

Options	Resource Class	Notes
(None)	RUC_WR_ALU	Dest/dst_bnk should point at bit bucket

Operations to read **aluXm** internal state:

Options	Resource Class	Notes
(None)	RUC_SP_MUL	src registers specified are ignored

Operations to write **aluXa** internal state:

Options	Resource Class	Notes
(None)	RUC_SP_ALU	src registers specified are ignored

Write operations below take their argument from **src1**.

- **put_shift_count_register src1 0x70**
Shift Count register write.
- **get_shift_count_register dest 0x71**
Shift Count register read.
- **fa_put_interrupt_enable_register src1 0x74**
Interrupt enable register write (to **aluXa**).
- **fa_get_interrupt_enable_register dest 0x75**
Interrupt enable register read (from **aluXa**).
- **fm_put_interrupt_enable_register src1 0xbe**
Interrupt enable register write (to **aluXm**).
- **fm_get_interrupt_enable_register dest 0xbb**
Interrupt enable register read (from **aluXm**).
- **fa_put_floating_mode_register src1 0x76**
Mode register write (to **aluXa**).
- **fa_get_floating_mode_register dest 0x77**
Mode register read (from **aluXa**).
- **fm_put_floating_mode_register src1 0xae**
Mode register write (to **aluXm**).
- **fm_get_floating_mode_register dest 0xbd**
Mode register read (from **aluXm**).

3.2.3 FALU0I AND FALU1I INTEGER OPERATIONS

The operations available on `falu0i` and `falu1i` are a subset of the opcodes available on the I board's `ialu0` and `ialu1`. Simple 32-bit integer operations use one or two 32-bit operands and produce a 32-bit result. Pair integer operations execute on two or four 32-bit operands to produce two 32-bit results. A pair operation specifying register 8 as an operand reads register 8 and 9. A pair operation specifying register 9 as an operand reads register 9 twice.

Both simple and pair integer operations are available in delayed mode. "Fast" versions of the integer operations are not available -- they're already low latency one beat operations.

All opcodes in this section also have the `Fpop` bit cleared.

The primary reasons `falu0i` and `falu1i` are included are to 1) Check and generate parity for the register files. 2) Provide a select opcode. 3) Provide a fast move path. Data for `GC_WRITE` operations passes through `falu0i`. The rest of the opcodes come "for free." They are documented in detail here (ie, the relevant chapters in the architecture specification for the I board are inserted) and special cases are noted.

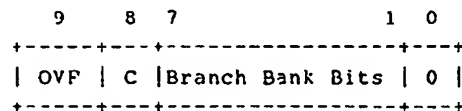
Integer operations fall into one of the following resource usage categories:

Simple Integer Operations		
Options	Resource Class	Notes
(None)	RUC_IALU	
DELAY	RUC_DLY_IALU	
Pair Integer Operations		
Options	Resource Class	Notes
(None)	RUC_PM_IALU	
DELAY	RUC_PM_DLY_IALU	

3.2.3.1 State register operations.

Each integer ALU contains a local integer state register. Certain operations manipulate one or more of these bits. The following two operations provide for loading and unloading the entire integer state register.

The format of this status is as follows:



Note that the only branch bank bit relevant to the F board is bit 1, used by the select opcode.

Writes to branch bank/state bit 0 are ignored. It always reads as zero.

- `gizr DEST 28`
Get Integer State Register.

The internal state registers of the IALU are written into `DEST`. `OVF` is the sticky overflow bit. It indicates that an overflow has been detected since the last time the bit was cleared (with a `PPSW`). This enables overflows to be polled for even while they're

masked from causing traps. C is the Carry bit.

All other bit positions are undefined.

This opcode is not available in pair mode.

- **pisr SRC2 2A**
Put Integer State Register

The value accessed by SRC2 is loaded into the internal state registers of the IALU. Its format is as in gisr SRC1 is ignored. DEST should be the bit bucket.

Undefined bit positions must be zero.

This opcode is not available in pair mode.

3.2.3.2 Integer Arithmetic Operations, signed operations

SRC1 and SRC2 are treated as 32-bit signed quantities. SRC1 and SRC2 are combined, and the 32 least significant bits of the result are written to DEST. Overflow is signaled if SRC1 op SRC2 does not lie within the range $-2147483648 \leq (\text{SRC1 op SRC2}) \leq 2147483647$.

On the F board, the behavior of the integer ALU's carry bit may be unpredictable.

- **add.s32 DEST, SRC1, SRC2 50**
DEST := SRC1 + SRC2
- **padd.s32 DEST, SRC1, SRC2 51**
Pair DEST := SRC1 + SRC2
- **add1.s32 DEST, SRC1, SRC2 52**
DEST := SRC1 + SRC2 + 1
- **padd1.s32 DEST, SRC1, SRC2 53**
Pair DEST := SRC1 + SRC2 + 1
- **addc.s32 DEST, SRC1, SRC2 54**
DEST := SRC1 + SRC2 + C
- **padde.s32 DEST, SRC1, SRC2 55**
Pair DEST := SRC1 + SRC2 + C
- **sub.s32 DEST, SRC1, SRC2 42**
DEST := SRC1 - SRC2
- **psub.s32 DEST, SRC1, SRC2 43**
Pair DEST := SRC1 - SRC2
- **sub1.s32 DEST, SRC1, SRC2 40**
DEST := SRC1 - SRC2 - 1
- **psub1.s32 DEST, SRC1, SRC2 41**
Pair DEST := SRC1 - SRC2 - 1
- **subc.s32 DEST, SRC1, SRC2 44**
DEST := SRC1 - SRC2 - 1 + C
The carry bit may act unpredictably.

- **psubc.s32** DEST, SRC1, SRC2 45
Pair DEST := SRC1 - SRC2 - 1 + C
The carry bit may act unpredictably.

3.2.3.3 Integer Arithmetic, Unsigned

These opcodes never signal overflow, but otherwise produce identical results to their “signed” counterparts. The carry (C-bit) is updated, *again, perhaps unpredictably.*

- **add.u32** DEST, SRC1, SRC2 58
DEST := SRC1 + SRC2
- **padd.u32** DEST, SRC1, SRC2 59
Pair DEST := SRC1 + SRC2
- **add1.u32** DEST, SRC1, SRC2 5A
DEST := SRC1 + SRC2 + 1
- **padd1.u32** DEST, SRC1, SRC2 5B
Pair DEST := SRC1 + SRC2 + 1
- **addc.u32** DEST, SRC1, SRC2 5C
DEST := SRC1 + SRC2 + C
Carry may act in unpredictable ways.
- **paddc.u32** DEST, SRC1, SRC2 5D
Pair DEST := SRC1 + SRC2 + C
Carry may act in unpredictable ways.
- **adds2.u32** DEST, SRC1, SRC2 6A
DEST := 2*SRC1 + SRC2
- **padds2.u32** DEST, SRC1, SRC2 6B
Pair DEST := 2*SRC1 + SRC2
- **adds4.u32** DEST, SRC1, SRC2 6C
DEST := 4*SRC1 + SRC2
- **padds4.u32** DEST, SRC1, SRC2 6D
Pair DEST := 4*SRC1 + SRC2
- **adds8.u32** DEST, SRC1, SRC2 6E
DEST := 8*SRC1 + SRC2
- **padds8.u32** DEST, SRC1, SRC2 6F
Pair DEST := 8*SRC1 + SRC2
- **sub.u32** DEST, SRC1, SRC2 4A
DEST := SRC1 - SRC2
- **psub.u32** DEST, SRC1, SRC2 4B
Pair DEST := SRC1 - SRC2
- **sub1.u32** DEST, SRC1, SRC2 48
DEST := SRC1 - SRC2 - 1
- **psub1.u32** DEST, SRC1, SRC2 49
Pair DEST := SRC1 - SRC2 - 1

- `subc.u32 DEST, SRC1, SRC2 40`
 $DEST := SRC1 - SRC2 - 1 + C$
Carry may act in unpredictable ways.
- `psubc.u32 DEST, SRC1, SRC2 41?`
 $Pair\ DEST := SRC1 - SRC2 - 1 + C$
Carry may act in unpredictable ways.

3.2.3.4 Shift Operations

- `lsrz.32 DEST, SRC1, SRC2 86`
- `plsrz.32 DEST, SRC1, SRC2 87`
 Logical Shift left, bitReverse, Zero fill

SRC1 is left shifted by the number of places specified by SRC2, with zeroes brought in on the right; then the bit ordering is reversed. If SRC2 is zero, no shifting is done, and the operation is a pure bitreverse (bit 0 becomes bit 31, etc.) This operation is defined only on the range $0 \leq SRC2 \leq 31$. Overflow is ignored.

- `lsrs.32 DEST, SRC1, SRC2 82`
- `plsr.32 DEST, SRC1, SRC2 83`
 Logical Shift left, bitReverse, Sign fill

SRC1 is left shifted by the number of places specified by SRC2, with the sign (bit 0 of SRC1) brought in on the right; then the bit ordering is reversed. If SRC2 is zero, no shifting is done, and the operation is a pure bitreverse (bit 0 becomes bit 31, etc.) This operation is defined only on the range $0 \leq SRC2 \leq 31$. Overflow is ignored. This operation is provided to allow arithmetic shift right: bitreverse the operand using `lsrz.32` with a zero shift count, then `lsrs.32` the result, with the shift count being the number of places to arithmetically shift right. For a logical right shift, use `lsrz.32` in step two.

- `lsz.32 DEST, SRC1, SRC2 84`
- `plsz.32 DEST, SRC1, SRC2 85`
 Logical Shift Left, Zero Fill

SRC1 is left shifted by the number of places specified by SRC2. Zeroes are brought in on the right. If SRC2 is zero, no shifting is done (DEST is set to SRC1). This operation is defined only on the range $0 \leq SRC2 \leq 31$. Overflow is ignored.

3.2.3.5 Extract Operations

The low order 2 bits of SRC2 are used to select a byte or halfword in SRC1. The most significant byte is number 2#00, as in main memory. The LSB of the selected field is shifted to the LSB of dest. Halfword Extracts select bits 31:16 or 15:0, only.

- `ext.u8 70`
- `pext.u8 71`
 $sel := SRC2 \& 2\#11; DEST := rshift(bytesel(SRC1, sel), 8*(3-sel));$ no sign extension

- **ext.s8** 74
- **pext.s8** 75
sel := SRC2 & 2#11; DEST := sxt(rshft(bytesel(SRC1,sel).8*(3-sel)); byte is sign-extended
- **ext.u16** 72
- **pext.u16** 73
sel := rshft(SRC2 & 2#10,1); DEST := rshft(hwordsel(SRC1,sel), 16*(1-sel)); no sign extension
- **ext.s16** 76
- **pext.s16** 77
sel := rshft(SRC2 & 2#10,1); DEST := sxt(rshft(hwordsel(SRC1,sel), 16*(1-sel))); sign extended

3.2.3.6 Select Operation

If branch bank register 1 = 0 then DEST is set to the value accessed by SRC1, else DEST is set to the value accessed by SRC2.

- **slet.32** DEST, SRC1, SRC2 2C
Select
- **pslet.32** DEST, SRC1, SRC2 2D
Pair Select

3.2.3.7 Merge Operations

These four opcodes have extra resource usage; they consume the SRC2 field for the other ALU in the current pair (e.g., if a mrg.u8 is done on cl2 falu0i, the cl2 falu1 SRC2 field is required). They get their SRC3 operands from the SRC2 leg of the other ALU of the current pair. SRC1 contains a 32 bit operand, SRC2 contains an 8 or 16 bit operand, SRC3 selects the field to be replaced. The least significant byte or halfword of SRC2 replaces the field in SRC1 selected by SRC3 & 2#11. The result is written to DEST. For the signed (.s) opcodes, overflow is signaled if SRC2 cannot fit into the field it is being merged into. For the .8 opcodes, SRC2 checked for range -128 <= SRC2 <= 127; for the .16, the range is -32768 <= SRC2 <= 32767. Overflow is signaled if it is out of range. Halfword Merges replace bits 31:16 or 15:0, only.

- **mrg.u8** DEST, SRC1, SRC2, (SRC3) 78
- **pmrg.u8** DEST, SRC1, SRC2, (SRC3) 79
DEST := bytemerge(SRC1, SRC2, SRC3&2#11); no overflow check
- **mrg.s8** DEST, SRC1, SRC2, (SRC3) 7C
- **pmrg.s8** DEST, SRC1, SRC2, (SRC3) 7D
DEST := bytemerge(SRC1, SRC2, SRC3&2#11); overflow is checked
- **mrg.u16** DEST, SRC1, SRC2, (SRC3) 7A
- **pmrg.u16** DEST, SRC1, SRC2, (SRC3) 7B
sel := rshft(SRC3 & 2#10, 1); DEST := hwordmerge(SRC1, SRC2, sel); no overflow check

- **ext.s8 74**
- **pext.s8 75**
sel := SRC2 & 2#11; DEST := sxt(rshft(bytesel(SRC1,sel),8*(3-sel))); byte is sign-extended
- **ext.u16 72**
- **pext.u16 73**
sel := rshft(SRC2 & 2#10,1); DEST := rshft(hwordsel(SRC1,sel), 16*(1-sel)); no sign extension
- **ext.s16 76**
- **pext.s16 77**
sel := rshft(SRC2 & 2#10,1); DEST := sxt(rshft(hwordsel(SRC1,sel), 16*(1-sel))); sign extended

3.2.3.6 Select Operation

If branch bank register 1 = 0 then DEST is set to the value accessed by SRC1, else DEST is set to the value accessed by SRC2.

- **slet.32 DEST, SRC1, SRC2 2C**
Select
- **pslet.32 DEST, SRC1, SRC2 2D**
Pair Select

3.2.3.7 Merge Operations

These four opcodes have extra resource usage; they consume the SRC2 field for the other ALU in the current pair (e.g., if a mrg.u8 is done on cl2 falu0i, the cl2 falu1 SRC2 field is required). They get their SRC3 operands from the SRC2 leg of the other ALU of the current pair. SRC1 contains a 32 bit operand, SRC2 contains an 8 or 16 bit operand, SRC3 selects the field to be replaced. The least significant byte or halfword of SRC2 replaces the field in SRC1 selected by SRC3 & 2#11. The result is written to DEST. For the signed (.s) opcodes, overflow is signaled if SRC2 cannot fit into the field it is being merged into. For the .8 opcodes, SRC2 checked for range -128 <= SRC2 <= 127; for the .16, the range is -32768 <= SRC2 <= 32767. Overflow is signaled if it is out of range. Halfword Merges replace bits 31:16 or 15:0, only.

- **mrg.u8 DEST, SRC1, SRC2, (SRC3) 78**
- **pmrg.u8 DEST, SRC1, SRC2, (SRC3) 79**
DEST := bytemerge(SRC1, SRC2, SRC3&2#11); no overflow check
- **mrg.s8 DEST, SRC1, SRC2, (SRC3) 7C**
- **pmrg.s8 DEST, SRC1, SRC2, (SRC3) 7D**
DEST := bytemerge(SRC1, SRC2, SRC3&2#11); overflow is checked
- **mrg.u16 DEST, SRC1, SRC2, (SRC3) 7A**
- **pmrg.u16 DEST, SRC1, SRC2, (SRC3) 7B**
sel := rshft(SRC3 & 2#10, 1); DEST := hwordmerge(SRC1, SRC2, sel); no overflow check

- **mrg.s16** DEST, SRC1, SRC2, (SRC3) 7E
- **pmrg.s16** DEST, SRC1, SRC2, (SRC3) 7F
sel := rshft(SRC3 & 2#10, 1); DEST := hwordmerge(SRC1, SRC2, sel); overflow is checked

3.2.3.8 Logical Operations

SRC1, SRC2 are “logically normalized”; that is, for each 32-bit operand, a 1-bit value is set to 1 (true) if any of the 32 bits of the operand is set. The boolean operation is performed, and the 1-bit result is right-aligned in a 32-bit field, the rest of which is zeroed.

- **and.32** 0x02
- **pand.32** 0x03
DEST := lnm(SRC1) & lnm(SRC2)
- **and_sli.32** 0x12
- **pand_sli.32** 0x13
DEST := NOT lnm(SRC1) & lnm(SRC2)
- **and_s2i.32** 0x0A
- **pand_s2i.32** 0x0B
DEST := lnm(SRC1) & NOT lnm(SRC2)
- **and_s1s2i.32** 0x1A
- **pand_s1s2i.32** 0x1B
DEST := NOT lnm(SRC1) & NOT lnm(SRC2)
- **or.32** 0x04
- **por.32** 0x05
DEST := lnm(SRC1) ! lnm(SRC2)
- **or_sli.32** 0x14
- **por_sli.32** 0x15
DEST := NOT lnm(SRC1) ! lnm(SRC2)
- **or_s2i.32** 0x0C
- **or_s2i.32** 0x0D
DEST := lnm(SRC1) ! NOT lnm(SRC2)
- **or_s1s2i.32** 0x1C
- **or_s1s2i.32** 0x1D
DEST := NOT lnm(SRC1) ! NOT lnm(SRC2)
- **eqv.32** 0x0E
- **eqv.32** 0x0F
DEST := lnm(SRC1) = lnm(SRC2)
- **neqv.32** 0x06
- **neqv.32** 0x07
DEST := not(lnm(SRC1) = lnm(SRC2))

3.2.3.9 Bitwise Logical Operations

SRC1, SRC2 are treated as 32-bit boolean bit strings. The two boolean string values are combined, and the 32-bit result string is written to DEST. If an I or F branch bank is specified as the destination, the LSB of the result is written.

- **band.32 0x32**
- **pband.32 0x33**
DEST := SRC1 & SRC2
- **band_s2i.32 0x3A**
- **pband_s2i.32 0x3B**
DEST := SRC1 & bitwise_not(SRC2)
- **bor.32 0x34**
- **pbor.32 0x35**
DEST := SRC1 ! SRC2
- **bor_s2i.32 0x3C**
- **pbor_s2i.32 0x3d**
DEST := SRC1 ! bitwise_not(SRC2)
- **bxor.32 0x36**
- **pbxor.32 0x37**
DEST := SRC1 xor SRC2
- **bxnor.32 0x3E**
- **pbxnor.32 0x3f**
DEST := bitwise_not(SRC1 xor SRC2)

3.2.3.10 Compare Operations

These opcodes do signed and unsigned compares of 32-bit integers. IF SRC1 *relop* SRC2 is true, then DEST = 1, else DEST = 0. DEST_BANK can point at normal register banks or a branch bank; result is always logically normalized.

- **ceq.u32 0xA4**
- **pceq.u32 0xA5**
DEST := SRC1 = SRC2
- **cne.u32 0xA0**
- **pcne.u32 0xA1**
DEST := SRC1 <> SRC2
- **clt.u32 0xAE**
- **pclt.u32 0xAF**
DEST := SRC1 <_u SRC2
- **cge.u32 0xAA**

- **pcge.u32 0xAB**
DEST := SRC1 >=u SRC2
- **cle.u32 0xA8**
- **pcle.u32 0xA9**
DEST := SRC1 <=u SRC2
- **cgt.u32 0xAC**
- **pcgt.u32 0xAD**
DEST := SRC1 >u SRC2
- **clt.s32 BE**
- **pclt.s32 BF**
DEST := SRC1 < SRC2
- **cge.s32 BA**
- **pcge.s32 BB**
DEST := SRC1 => SRC2
- **cle.s32 B8**
- **pcle.s32 B9**
DEST := SRC1 <= SRC2
- **cgt.s32 BC**
- **pcgt.s32 BD**
DEST := SRC1 > SRC2
- **ceqsgn.s32 DEST, SRC1, SRC2 B4**
- **pceqsgn.s32 DEST, SRC1, SRC2 B5**
Compare Equal Signs

SRC1 and SRC2 are treated as signed 32 bit integers. F ((SRC1 >= 0) AND (SRC2 >= 0)) OR ((SRC1 < 0) AND (SRC2 < 0)), then DEST = 1. else DEST = 0. DEST_BANK can be a normal or branch bank. If DEST is in a normal bank, it will be written with a logically normalized value.

- **cneqsgn.s32 DEST, SRC1, SRC2 B2**
- **pcneqsgn.s32 DEST, SRC1, SRC2 B3**
Compare Not Equal Signs

SRC1 and SRC2 are treated as signed 32 bit integers. IF ((SRC1 >= 0) AND (SRC2 < 0)) OR ((SRC1 < 0) AND (SRC2 >= 0)), then DEST = 1, else DEST = 0. DEST_BANK can be a normal or branch bank. If DEST is in a normal bank, it will be written with a logically normalized value.

- **matchc DEST, SRC1, SRC2 A6**
- **pmatchc DEST, SRC1, SRC2 A7**
Compare Byte-by-Byte. Unsigned

SRC1 must be a zero register. Each byte of SRC2 is compared with zero. If no bytes are zero, DEST = 7. If the most significant byte is zero, DEST = 0; if the next-most, DEST=1; if the next-most, DEST = 2; if least significant are zero, DEST = 3. If the destination of the operation is the branch bank, no logical normalization occurs.

3.2.4 SPECIAL OPCODES

All opcodes in this section also have the Fpop bit set.

3.2.4.1 Floating Status Register Manipulation

The following opcodes are provided to manipulate the floating status register.

- **put_floating_status_register SRC2 0xce**
External floating status register write. The lower 16 bits of the 32-bit value supplied by SRC2 is written into the floating status register. Falul only. Note that output resources are required even if the destination is the bit bucket. If the destination is not the bit bucket, it is written in beat 2 with the former value in the register. Special restriction: Dst_bnk may not be 4-7 for this opcode (ie, the result cannot be sent over the FL bus.)

Resource	#Used	When	Notes
falul_src	1	0	
falul_out	1	1	-- Tail --
falul_out	1	2	-- Tail --

- **get_floating_status_register DEST 0xcf**
External floating status register read. The 16-bit FSR is written to DEST. (Upper, unassigned bits read as zero). Note the extra output usage in beat 1; the result of this write is undefined – the final value is written in beat 2. Special restriction: Dst_bnk may not be 4-7 for this opcode (ie, the result cannot be sent over an FL bus). falul only.

Resource	#Used	When	Notes
falul_out	1	1	-- Tail --
falul_out	1	2	-- Tail --

3.2.4.2 Explicit Sanitizers

The mv_check opcodes are used to check a value for Infinity or NaN while possibly moving it to another location.

Options	Resource Class	Notes
(None)	RUC_SP_ALU	
DELAY	RUC_SP_DLY_ALU	
FAST	RUC_SP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_SP_DLY_FST_ALU	Requires fast ALU

- **mv_check.f32 DEST, SRC1 0x00**
The 32-bit SRC1 is moved to DEST, and the value is checked for Infinity or NaN. If Infinity is detected, an overflow trap is signalled. If NaN is detected, an invalid operation trap is signalled. *These exceptions are not disabled by any mode registers; they will unconditionally signal a floating exception if NaN or Infinity is seen..*

Options	Resource Class	Notes
(None)	RUC_DP_ALU	
DELAY	RUC_DP_DLY_ALU	
FAST	RUC_DP_FST_ALU	Requires fast ALU
DELAY, FAST	RUC_DP_DLY_FST_ALU	Requires fast ALU

- **mv_check.f64 DEST, SRC1 0x01**
The 64-bit SRC1 is moved to DEST, and the value is checked for Infinity or NaN. If Infinity is detected, an overflow trap is signalled. If NaN is detected, an invalid operation trap is signalled. *These exceptions are not disabled by any mode registers: they will unconditionally signal a floating exception if NaN or Infinity is seen..*

3.2.5 NO-OP

Options	Resource Class	Notes
(None)	RUC_NOP	

- **NOP 0x00**
All fields in the operation must be zero. No output or core resources used.

3.2.6 GLOBAL CONTROLLER OPERATIONS

GCOPs are available only on falu0, and block the initiation of a floating point or integer operation on falu0.

Initially, rather than listing the GCOPs individually, the following algorithm is given to convert /200 GCOPs to Janus GCOPs:

- GCOP, DEST, SRC1, SRC2, DST_BNK, and BRDCST have the same values for Janus as for Saturn.
- FPOP=0 in the Janus opcode.
- If the operation is an MNOP, then OPCODE=0xff, else
- If Saturn's IRO==0 then Janus's OPCODE=0xc0, else
- If Saturn's IRO==1 and 64==0 then Janus's OPCODE=0x24 and DELAY=1.
- If Saturn's IRO==1 and 64==1 then Janus's OPCODE=0x25 and DELAY=0.
- FPOP is always zero for GCOPs.

CHAPTER 4 RESOURCES

4.1 RESOURCES

Machine resources must be considered by the compiler or hand coder when scheduling F unit operations. These sections describe machine resources used by or visible to the F unit.

Programs overusing or misusing F resources will run incorrectly, and often non-deterministically. Non-deterministic operation results from asynchronous interactions of the program with interrupts and cache misses – an interrupt may allow one operation to drain before another is issued, freeing a resource for the next operation.

In most cases, resource overuse will not be directly detected by the hardware, and the program or process will continue execution.

4.2 AVAILABLE RESOURCES

The following resources are replicated on each cluster. Only resources pertaining to the F unit are listed.

Resource	#Available	Description
faluo_src	2/cl	faluo src1/src2 read port
falua_core	1/cl	Internal computational core of the BIT alu on faluo
falum_core	1/cl	Internal computational core of the BIT mpy on faluo
falui_core	1/cl	Internal logic of the MF ialu on faluo
faluo_out	1/cl	Output bus for faluo functional units
falul_src1	1/cl	faluo src1 read port
falul_src2	1/cl	faluo src2 read port
falula_core	1/cl	Internal computational core of the BIT alu on faluo
falulm_core	1/cl	Internal computational core of the BIT mpy on faluo
faluli_core	1/cl	Internal logic of the MF ialu on faluo
falul_out	1/cl	Output bus for faluo functional units
f0_bb_wr	1/cl	faluo branch bank write
f1_bb_wr	1/cl	falul branch bank write
f_rf_write	2/subbank/cl	Register file write resources
f_rf_alu_wr	1/subbank/cl	Register file alu write resources
srf_write	2/cl	Store register file write resources
if_bus	2/cl	I<->F buses.
faluo_flag	1/cl	faluo flag output (** see discussion below **)
falul_flag	1/cl	falul flag output (** see discussion below **)
faluo_spdest	1/cl	faluo special destination resource
falul_spdest	1/cl	falul special destination resource

These resources are shared by all F units in the machine.

Resource	#Available	Description
----------	------------	-------------

fl_bus	4	FL buses
gcop	1	GC operation bus

In addition to managing F resources, the following I board resources must also be managed when scheduling F operations:

Resource	#Available	Description
i0_bb_write	1/cl	ialu0 branch bank write for local I board
i_rf_write	2/subbank/cl	register file write resource for local I board.

Other I board resources exist but are not affected by F operations scheduling.

4.3 REGISTER FILE RESOURCES

When an operation targets a register in the register file, the value in that register must be considered undefined. Other operations cannot read the old or new value in the target register until the first operation completes.

This restriction is necessary because an interrupt can occur at any time and will drain the pipelines. The process/program sees this as operations that occasionally and unpredictably complete instantaneously.

4.4 FALU0_FLAG AND FALU1_FLAG RESOURCES

The faluX_flag resource models the sharing of the floating point exception flag bus between falu0a and falu0m and between falu1a and falu1m. Floating point comparison operations use the flag bus in a manner which restricts particular combinations of FP compare and FP multiply.

If correct floating point exception generation is required, the faluX_flag resource must be considered when scheduling instructions.

If correct floating point exception behavior is not required (when the board is operating in "NOTRAP" mode) the faluX_flag resources can be ignored. If the faluX_flag resource is oversubscribed, the hardware will generate correct results, but may generate incorrect exceptions.

4.5 FALUX_SPDEST RESOURCES

The faluX_spdest resource models a control pipeline restriction that limits how close particular division and square root operations can follow one another.

4.6 EXAMPLE

The Janus F board operations can be classified by their resource usage. The "resource usage classes" listed below describe how machine resources are used as we initiate operations -- the "head" of the operation pipeline.

When an operation completes, the result is written in the manner directed by the `Dest`, `Dst_Bnk`, and `BrdCst` fields in the opcode. The resources used during these tails of operations are listed later. The position of the tail relative to the initiation of the operation is indicated by the "tail" entry in the resource usage class descriptions.

For example, given the operation:

```
instr
    cl0 falu01 mpy.f64 fb2.r32,r2,r4;
```

find the `mpy.f64` operation in the "Double Precision Multiply Operations" section in the "Operations" chapter of this section, and refer to the table there:

Options	Resource Class	Notes
(None)	RUC_DP_MUL	
DELAY	RUC_DP_DLY_MUL	
FAST	RUC_DP_FST_MUL	
DELAY,FAST	RUC_DP_DLY_FST_MUL	

Since the multiplication is being issued with the DELAY option (i.e. `falu01`), the operation resource use would be described by `RUC_DP_DLY_MUL`. In the list below, `RUC_DP_DLY_MUL` is described with the following table:

Resource	#Used	When	Notes
<code>faluX_src</code>	2	1	
<code>faluX_src</code>	2	2	
<code>faluXm_core</code>	1	3	
<code>faluX_out</code>	1	4	-- Tail --
<code>faluX_out</code>	1	5	-- Tail --

Since the destination specified is a non local cluster (`Dst_bnk == 110`), the tail resource usage characteristics are described with the following table.

`Dst_bnk == 1XX`

Resource	#Used	When	Notes
<code>faluX_out</code>	1	0	
<code>fl_bus</code>	1	0	
<code>f_rf_write</code>	1	1	(<code>Dest < 32</code>) ==> Sub bank 0, else Sub bank 1

So the resource usage for the opcode is as follows:

Resource use for

```
instr
    cl0 falu01 mpy.f64 fb2.r32,r2,r4;
```

Beat	Resource	Cluster	Notes
0	-- None --		Delayed opcode
1	<code>falu0_src (X2)</code>	0	Read MSWs of operands into FMUL
2	<code>falu0_src (X2)</code>	0	Read LSWs of operands into FMUL, start operation

3	falu0m_core	0	Perform the multiplication
4	falu0_out	0	Unload MSW of result from FMUL
	fl_bus	n/a	Transfer MSW to cluster 2
5	f_rf_write	2	Write MSW result into cl2 rf, sub bank 1
	falu0_out	0	Unload LSW of result from FMUL
	fl_bus	n/a	Transfer LSW to cluster 2
6	f_rf_write	2	Write LSW result into cl2 rf, sub bank 1

The next operation would start in beat 2. Note that resource rules prohibit issuing a single precision multiply in beat 2 because doing so would overuse falu0m_core and falu0_out resources.

4.7 RESOURCE USAGE CLASSES

Resource usage mnemonics (in parenthesis, after full name of resource usage class) are recommended for use in software and programmable logic sources.

1. No op (RUC_NOP)

Resource	#Used	When	Notes
faluX_instr 1	0		
faluX_instr 1	1		

2. Regular Double Precision, ALU (RUC_DP_ALU)

Resource	#Used	When	Notes
faluX_instr 1	0		
faluX_instr 1	1		
faluX_src 2	0		
faluX_src 2	1		
faluXa_core 1	2		
faluX_flag 1	3		
faluX_flag 1	4		
faluX_out 1	3		-- Tail --
faluX_out 1	4		-- Tail --

3. Regular Double Precision, MUL (RUC_DP_MUL)

Resource	#Used	When	Notes
faluX_instr 1	1	0	
faluX_instr 1	1	1	
faluX_src 2	0		
faluX_src 2	1		
faluXm_core 1	2		
faluX_flag 1	3		
faluX_flag 1	4		
faluX_out 1	3		-- Tail --
faluX_out 1	4		-- Tail --

4. Delayed Regular Double Precision, ALU (RUC_DP_DLY_ALU)

Resource	#Used	When	Notes
faluX_instr 1	1	0	
faluX_instr 1	1	1	

falux_src	2	1	
falux_src	2	2	
faluxa_core	1	3	
falux_flag	1	4	
falux_flag	1	5	
falux_out	1	4	-- Tail --
falux_out	1	5	-- Tail --

5. Delayed Regular Double Precision, MUL (RUC_DP_DLY_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	3	
falux_flag	1	4	
falux_flag	1	5	
falux_out	1	4	-- Tail --
falux_out	1	5	-- Tail --

6. Fast Double Precision, ALU (RUC_DP_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	1	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

7. Fast Double Precision, MUL (RUC_DP_FST_MUL)

*** NOT AVAILABLE ***

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	1	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

8. Delayed Fast Double Precision, ALU (RUC_DP_DLY_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	

falux_src	2	2	
faluxa_core	1	2	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

9. Delayed Fast Double Precision, MUL (RUC_DP_DLY_FST_MUL)

*** NOT AVAILABLE ***

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	2	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

10. Regular Single Precision, ALU (RUC_SP_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxa_core	1	1	
falux_flag	1	2	
falux_out	1	2	-- Tail --

11. Regular Single Precision, MUL (RUC_SP_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxm_core	1	1	
falux_flag	1	2	
falux_out	1	2	-- Tail --

12. Delayed Regular Single Precision, ALU (RUC_SP_DLY_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxa_core	1	2	
falux_flag	1	3	
falux_out	1	3	-- Tail --

13. Delayed Regular Single Precision, MUL (RUC_SP_DLY_MUL)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	1	
faluXm_core	1	2	
faluX_flag	1	3	
faluX_out	1	3	-- Tail --

14. Fast Single Precision, ALU (RUC_SP_FST_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	0	
faluXa_core	1	0	
faluX_flag	1	1	
faluX_out	1	1	-- Tail --

15. Fast Single Precision, MUL (RUC_SP_FST_MUL)

*** NOT AVAILABLE ***
Hardware resources previously
used by this resource usage
class are now being used by MAC.

16. Delayed Fast Single Precision, ALU (RUC_SP_DLY_FST_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	1	
faluXa_core	1	1	
faluX_flag	1	2	
faluX_out	1	2	-- Tail --

17. Delayed Fast Single Precision, MUL (RUC_SP_DLY_FST_MUL)

*** NOT AVAILABLE ***

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	1	
faluXm_core	1	1	
faluX_flag	1	2	
faluX_out	1	2	-- Tail --

18. Pair Mode Normal, ALU (RUC_PM_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	0	

falux_src	2	1	
faluxa_core	1	1	
faluxa_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

19. Pair Mode Normal, MUL (RUC_PM_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxm_core	1	1	
faluxm_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

20. Delayed Pair Mode, ALU (RUC_PM_DLY_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	2	
faluxa_core	1	3	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

21. Pair Mode Normal, MUL (RUC_PM_DLY_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	2	
faluxm_core	1	3	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

22. Fast Pair Mode, ALU (RUC_PM_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	

falux_src	2	0	
falux_src	2	1	
faluxa_core	1	0	
faluxa_core	1	1	
falux_flag	1	1	
falux_flag	1	2	
falux_out	1	1	-- Tail --
falux_out	1	2	-- Tail --

23. Fast Pair Mode, MUL (RUC_PM_FST_MUL)

*** NOT AVAILABLE ***

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxm_core	1	0	
faluxm_core	1	1	
falux_flag	1	1	
falux_flag	1	2	
falux_out	1	1	-- Tail --
falux_out	1	2	-- Tail --

24. Delayed Fast Pair Mode, ALU (RUC_PM_DLY_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	1	
faluxa_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

25. Delayed Fast Pair Mode, MUL (RUC_PM_DLY_FST_MUL)

*** NOT AVAILABLE ***

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	1	
faluxm_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

26. Integer Normal Double (RUC_PM_IALU)

A pair operation more than a double operation.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxi_core	1	0	
faluxi_core	1	1	
falux_out	1	1	-- Tail --
falux_out	1	2	-- Tail --

27. Delayed Integer Double (RUC_PM_DLY_IALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxi_core	1	1	
faluxi_core	1	2	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

28. Integer Normal Single (RUC_IALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxi_core	1	0	
falux_out	1	1	-- Tail --

29. Integer Delayed Single (RUC_DLY_IALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxi_core	1	1	
falux_out	1	2	-- Tail --

30. Regular Double Precision Compare (RUC_DP_CMP)

Beat 3 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	2	
falux_flag	1	3	

falux_out	1	4	-- Tail --
-----------	---	---	------------

31. Delayed Double Precision Compare (RUC_DP_CMP_DLY)

Beat 4 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	3	
falux_flag	1	4	
falux_out	1	5	-- Tail --

32. Fast Double Precision Compare (RUC_DP_CMP_FST)

Beat 2 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	1	
falux_flag	1	2	
falux_out	1	3	-- Tail --

33. Delayed, Fast Double Precision Compare (RUC_DP_CMP_DLY_FST)

Beat 3 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	2	
falux_flag	1	3	
falux_out	1	4	-- Tail --

34. Regular Single Precision Compare (RUC_SP_CMP)

Beat 2 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxa_core	1	1	
falux_flag	1	2	
falux_out	1	3	-- Tail --

35. Delayed Single Precision Compare (RUC_SP_CMP_DLY)

Beat 3 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxa_core	1	2	
falux_flag	1	3	
falux_out	1	4	-- Tail --

36. Fast Single Precision Compare (RUC_SP_CMP_FST)

Beat 1 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxa_core	1	0	
falux_flag	1	1	
falux_out	1	2	-- Tail --

37. Delayed Fast Single Precision Compare (RUC_SP_CMP_DLY_FST)

Beat 2 is used to convert BIT output status to logically normalized compare result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxa_core	1	1	
falux_flag	1	2	
falux_out	1	3	-- Tail --

38. Regular Convert Single to Double, FALU (RUC_SP_DP_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxa_core	1	1	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

39. Delayed Convert Single to Double, FALU (RUC_SP_DP_DLY_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxa_core	1	2	

falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

40. Fast Convert Single to Double, FALU (RUC_SP_DP_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxa_core	1	0	
falux_flag	1	1	
falux_flag	1	2	
falux_out	1	1	-- Tail --
falux_out	1	2	-- Tail --

41. Delayed Fast Convert Single to Double, FALU (RUC_SP_DP_DLY_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxa_core	1	1	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

42. Regular Integer Multiply, FMUL (RUC_SP_DP_MUL)

Two 32 bit inputs, yields a 64 bit result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
faluxm_core	1	1	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

43. Delayed Integer Multiply, FMUL (RUC_SP_DP_DLY_MUL)

Two 32 bit inputs, yields a 64 bit result.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
faluxm_core	1	2	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	3	-- Tail --

```

    faluX_out    1      4      -- Tail --

```

44. Fast Integer Multiply, FMUL (RUC_SP_DP_FST_MUL)

Two 32 bit inputs, yields a 64 bit result.

```

    *** NOT AVAILABLE ***
    Hardware resources previously
    used by this resource usage
    class are now being used by
    MAC.

```

45. Delayed Fast Integer Multiply, FMUL (RUC_SP_DP_DLY_FST_MUL)

Two 32 bit inputs, yields a 64 bit result.

```

    *** NOT AVAILABLE ***

```

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	1	
faluXm_core	1	1	
faluX_flag	1	2	
faluX_flag	1	3	
faluX_out	1	2	-- Tail --
faluX_out	1	3	-- Tail --

46. Regular Convert Double to Single, FALU (RUC_DP_SP_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	0	
faluX_src	2	1	
faluXa_core	1	2	
faluX_flag	1	3	
faluX_out	1	3	

47. Delayed Convert Double to Single, FALU (RUC_DP_SP_DLY_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	
faluX_instr	1	1	
faluX_src	2	1	
faluX_src	2	2	
faluXa_core	1	3	
faluX_flag	1	4	
faluX_out	1	4	

48. Fast Convert Double to Single, FALU (RUC_DP_SP_FST_ALU)

Resource	#Used	When	Notes
faluX_instr	1	0	

falux_instr	1	1
falux_src	2	0
falux_src	2	1
faluxa_core	1	1
falux_flag	1	2
falux_out	1	2

49. Delayed Fast Convert Double to Single, FALU (RUC_DP_SP_DLY_FST_ALU)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	2	
falux_flag	1	3	
falux_out	1	3	

50. Double Square Root, Normal (RUC_DP_SQRT_MUL)

600 ns latency inside the part – need 10 x 65 ns beats.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	0	
falux_src	2	1	
faluxm_core	1	2	(1)
faluxm_core	1	3	(2)
faluxm_core	1	4	(3)
faluxm_core	1	5	(4)
faluxm_core	1	5	(5)
faluxm_core	1	6	(6)
faluxm_core	1	8	(7)
faluxm_core	1	9	(8)
faluxm_core	1	10	(9)
faluxm_core	1	11	(10)
falux_spdest	1	11	** Restricts when next div/sqrt can begin
falux_flag	1	12	
falux_flag	1	13	
falux_out	1	12	-- Tail --
falux_out	1	13	-- Tail --

51. Delayed Double Precision Square Root (RUC_DP_SQRT_DLY_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	3	(1)
faluxm_core	1	4	(2)

faluxm_core	1	5	(3)
faluxm_core	1	6	(4)
faluxm_core	1	7	(5)
faluxm_core	1	8	(6)
faluxm_core	1	9	(7)
faluxm_core	1	10	(8)
faluxm_core	1	11	(9)
faluxm_core	1	12	(10)
falux_spdest	1	11	** Restricts when next div/sqrt can begin
falux_flag	1	13	
falux_flag	1	14	
falux_out	1	13	-- Tail --
falux_out	1	14	-- Tail --

52. Regular Single Precision Square Root (RUC_SP_SQRT_MUL)

300 ns latency inside the part -- need 5 x 65 ns beats.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	0	
faluxm_core	1	1	(1)
faluxm_core	1	2	(2)
faluxm_core	1	3	(3)
faluxm_core	1	4	(4)
faluxm_core	1	5	(5)
falux_flag	1	6	
falux_out	1	6	-- Tail --

53. Delayed Single Precision Square Root (RUC_SP_SQRT_DLY_MUL)

300 ns latency inside the part -- need 5 x 65 ns beats.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	1	
faluxm_core	1	2	(1)
faluxm_core	1	3	(2)
faluxm_core	1	4	(3)
faluxm_core	1	5	(4)
faluxm_core	1	6	(5)
falux_spdest	1	5	** Restricts when next sqrt/div will begin
falux_flag	1	7	
falux_out	1	7	-- Tail --

54. Double Precision Divide (RUC_DP_DIV_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	

falux_spdest	1	1	
falux_src	2	0	
falux_src	2	1	
faluxm_core	1	2	(1)
faluxm_core	1	3	(2)
faluxm_core	1	4	(3)
faluxm_core	1	5	(4)
faluxm_core	1	6	(5)
falux_spdest	1	5	** Restricts when next sqrt/div will begin
falux_flag	1	7	
falux_flag	1	8	
falux_out	1	7	-- Tail --
falux_out	1	8	-- Tail --

55. Delayed Double Precision Divide (RUC_DP_DIV_DLY_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	1	
falux_src	2	2	
faluxm_core	1	3	(1)
faluxm_core	1	4	(2)
faluxm_core	1	5	(3)
faluxm_core	1	6	(4)
faluxm_core	1	7	(5)
falux_spdest	1	7	** Restricts when next div/sqrt can begin
falux_flag	1	8	
falux_flag	1	9	
falux_out	1	8	-- Tail --
falux_out	1	9	-- Tail --

56. Single Precision Divide (RUC_SP_DIV_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	0	
faluxm_core	1	1	(1)
faluxm_core	1	2	(2)
faluxm_core	1	3	(3)
faluxm_core	1	4	(4)
falux_spdest	1	3	** Restricts when next div/sqrt can begin
falux_flag	1	5	
falux_out	1	5	-- Tail --

57. Delayed Single Precision Divide (RUC_SP_DIV_DLY_MUL)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	

falux_src	2	1	
faluxm_core	1	2	(1)
faluxm_core	1	3	(2)
faluxm_core	1	4	(3)
faluxm_core	1	5	(4)
falux_flag	1	6	
falux_out	1	6	-- Tail --

58. Pair Compare (RUC_PM_CMP)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	1	
faluxa_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

59. Pair Compare (RUC_PM_CMP_DLY)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	1	
falux_src	2	2	
faluxa_core	1	2	
faluxa_core	1	3	
falux_flag	1	3	
falux_flag	1	4	
falux_out	1	4	-- Tail --
falux_out	1	5	-- Tail --

60. Pair Compare (RUC_PM_CMP_FST)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	0	
faluxa_core	1	1	
falux_flag	1	1	
falux_flag	1	2	
falux_out	1	2	-- Tail --
falux_out	1	3	-- Tail --

61. Pair Compare (RUC_PM_CMP_DLY_FST)

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	

falux_src	2	1	
falux_src	2	2	
faluxa_core	1	1	
faluxa_core	1	2	
falux_flag	1	2	
falux_flag	1	3	
falux_out	1	3	-- Tail --
falux_out	1	4	-- Tail --

62. Double Precision Multiply Accumulate Initialize (RUC_DP_MACINIT) Operation destinations fb0, fb1, fb2, and fb3 are illegal for this RUC.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	0	
falux_src	2	1	
faluxa_core	1	2	
faluxa_core	1	3	
faluxa_core	1	4	
faluxa_core	1	5	
falux_spdest	1	3	(only if isolated)
falux_flag	1	5	
falux_flag	1	6	
falux_out	1	3	
falux_out	1	4	
falux_out	1	5	(only if isolated)
falux_out	1	6	(only if isolated)
(None)		5	-- Tail (only if isolated)
(None)		6	-- Tail (only if isolated)

63. Double Precision Multiply Accumulate (RUC_DP_MAC) Operation destinations fb0, fb1, fb2, and fb3 are illegal for this RUC.

Resource	#Used	When	Notes
falux_instr	1	0	
falux_instr	1	1	
falux_spdest	1	1	
falux_src	2	0	
falux_src	2	1	
faluxm_core	1	2	
faluxm_core	1	3	
faluxa_core	1	4	
faluxa_core	1	5	
falux_spdest	1	3	(only used if isolated)
falux_flag	1	5	
falux_flag	1	6	
falux_out	1	3	
falux_out	1	4	
falux_out	1	5	(only if isolated)
falux_out	1	6	(only if isolated)
(None)		5	-- Tail (only if isolated)
(None)		6	-- Tail (only if isolated)

4.7.1 TAIL RESOURCE CLASSES

Bus and write resources are used to transfer results of operations to destination and write results. The destination bank specified at time of issue determines the resources and when they are used.

These resource tables are indexed from the beat that the result appeared on the `aluX_out` bus.

1. Bit Bucket (TAIL_BITBUCKET).

`Dest == 000000, Dst_bnk == 000`

No bus or write resourced used.

2. Local I branch bank (TAIL_IBB).

`000001 <= Dest <= 000111, Dst_bnk == 000`

Resource	#Used	When	Notes
<code>aluX_out</code>	1	0	
<code>i0_bb_write</code>	1	1	

3. F branch bank (TAIL_FBB_WR)

`(Dest == 100000) || (Dest == 100001), Dst_bnk == 000`

Resource	#Used	When	Notes
<code>aluX_out</code>	1	0	
<code>fx_bb_wr</code>	1	1	

A "select" operation can be performed on the result in beat 1.

4. Local Floating Register File (TAIL_LOCFRF)

`Dst_bnk == 001`

Resource	#Used	When	Notes
<code>aluX_out</code>	1	0	
<code>f_rf_write</code>	1	0	<code>(Dest < 32) ==> Sub bank 0, else Sub bank 1</code>
<code>f_rf_alu_wr</code>	1	0	<code>(Dest < 32) ==> Sub bank 0, else Sub bank 1</code>

5. Local Integer Bank (TAIL_LOCI RF)

`Dst_bnk == 010`

Resource	#Used	When	Notes
<code>aluX_out</code>	1	0	
<code>if_bus</code>	1	0	
<code>i_rf_write</code>	1	1	<code>(Dest < 32) ==> Sub bank 0, else Sub bank 1</code>

6. Local Store Register File (TAIL_LOCSRF)

`Dst_bnk == 011`

Resource	#Used	When	Notes
<code>aluX_out</code>	1	0	

```
srf_write      1      1
```

7. Remote Floating Register File (TAIL_REMFRR)

```
Dst_bnk == 1XX
```

Resource	#Used	When	Notes
faluX_out	1	0	
fl_bus	1	0	
f_rf_write	1	1	(Dest < 32) ==> Sub bank 0, else Sub bank 1

The FL bus transfer will use any available fl bus. Register file write ports are used on the addressed cluster.

CHAPTER 5

FLOATING EXCEPTIONS AND TRAPS

5.1 FLOATING EXCEPTIONS AND TRACE SCHEDULING

The desires for high performance parallel computing and complete IEEE support for floating exceptions and traps are opposed. Maximum floating point performance requires code optimizations, distancing the object code executed from the original source, and causing the executing object to deviate from the original program in its generation of exceptions. Restricting code transformations to those that are "IEEE-exception-invariant" limits optimization and performance.

Full IEEE trap support includes providing the user with the ability to specify a trap handler for any of the five exceptions. This trap handler is to be able to examine the operands and "wrapped" result of the exceptional operation, and to be able to return an arbitrary value as the result of the operation before continuing program execution. In many cases, aggressive code scheduling will blur the context of an operation, making it difficult or impossible to locate the operands and identify the operation for the trap handler. In addition, the overhead associated with invoking the trap handler will severely limit performance by orders of magnitude.

The hardware provides two modes of operation for floating exceptions: ALLTRAP, and DELAYTRAP, selected by the FSR_ALLTRAP_MODE bit in the floating status register.

5.2 ALLTRAP MODE

When ALLTRAP mode is set, the result of every floating point operation is checked for a floating exception, and sticky bits are set in the floating status as required. If the corresponding bit in the functional unit's interrupt enable register is set, the F board will also request a floating exception trap from the global controller. The global controller will detect the condition, interrupt the program, and cause a jump to trap code.

When in ALLTRAP mode, the following exception modes can be supported: Full IEEE Exception Support and Limited IEEE Exception Support.

5.2.1 FULL IEEE EXCEPTION SUPPORT

Programs running with full IEEE exception support will run with poor floating point performance.

- Compiler may not perform code transformations that modify IEEE exception behavior.
- Code generator cannot compact operations in a manner that would prevent the trap code from providing information to the user trap handler as required by IEEE 754, section 8.1:

- a. Which exception occurred on this operation;
 - b. The kind of operation that was being performed;
 - c. The destination's format;
 - d. In overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination's format; and
 - e. In invalid operation and divide by zero exceptions, the operand values.
- User code provides a trap handler, and sets interrupt mask word desired.
 - On an exception, system trap code gathers the required information above, invokes the user trap handler, and waits for it to terminate with a system call. It then substitutes the value returned by the trap handler, and continues program execution. To generate the "correctly rounded result," the trap code might be able to set the functional unit into a "wrapped result mode" (or normal operation could always occur in that mode).

5.2.2 LIMITED IEEE EXCEPTION SUPPORT

Limited IEEE Exception mode offers better performance, but does not allow the user to supply a trap handler.

- The Compiler still may not perform code transformations that modify IEEE exception behavior.
- Compiler operation scheduling can be more aggressive, reusing operand registers and pipelining operations. Still have to be careful about moving operations relative to splits and joins (these are not IEEE invariant operations).
- User code sets exception mode desired.
- On an unmasked exception, trap code interrupts the process, and provides information about exceptions that occurred.

5.3 DELAYTRAP MODE

When not in ALLTRAP mode, the F board operates in DELAYTRAP mode.

5.3.1 DELAYTRAP MODE

This mode offers good floating point exception detection, without incurring a large performance penalty. It should be the default floating exception mode. In delay trap mode, we only consider overflow and invalid operation exceptions to be serious.

- The compiler is much more aggressive when scheduling operations, scheduling them freely, with small regard for floating exception information. It depends upon NaNs and Infinities that result from intermediate, unchecked floating point operations to propagate through a chain of operations until special "sanitizing" operations are executed. Sanitizing operations implicitly or explicitly watch for Infinities or NaNs, and immediately generate an exception when they see one. Sanitizing operations may not move above splits.

- In theory, an operation's movement above a split is important because it can lead to spurious floating point exceptions. For example, if a division operation moved above a test that prevented its denominator from being zero, we would get divide by zero exceptions even though the program tried to prevent it. Postponing divide by zero exceptions until a sanitizing operation reacts to the division's result allows us to move the division as high as possible in the schedule without having to worry about the spurious exception.
- A value that has the potential of being an Infinity or NaN is considered a "dirty" value. Values loaded from memory are considered not dirty.

Any operation that can generate a NaN or Infinity is considered "sloppy." The output of a sloppy operation is a dirty result.

Some operations are neutral. They do not generate dirty results themselves, but if passed a dirty operand, produces a dirty result. An example of a neutral operation is SELECT.

Any dirty value must be sanitized before it is put into the store register file, because the value could be potentially stored to memory (and memory is not dirty).

Other operations require sanitized inputs. An operation requires a sanitized input if it would destroy a NaN or an Infinity from that input. For example, conversions from floating point to integer formats does not propagate NaN. Denominators for division operations must be sanitized, because they might not propagate an infinity.

Because floating point compare operations produce a boolean result, and do not propagate INF and NaN, they require sanitization of their inputs. Unfortunately, implementing compare operations so that they are implicitly sanitizing is not possible with the BIT parts, so operations to explicitly check values must be inserted before comparing them.

Any neutral or sloppy operation that targets its dirty result to the store register file becomes implicitly sanitizing (this is a hardware feature.) No extra explicit sanitizing operation is required. Note that this restricts these neutral and sloppy operations from moving above splits.

Operations converting floating point numbers to integers are implicitly sanitizing, to prevent the need for explicit sanitizing operations. (This also is a hardware feature).

The MV_CHECK opcodes are explicitly sanitizing. These opcodes check for NaN or Infinity as an operand and generate exceptions if either value is seen. (A free side effect of this operation can be to perform a move.) These operations are inserted when implicit sanitizing does not suffice, like when a division uses a dirty value as a denominator. Operations that target their results to the local integer file are also implicitly sanitized. For example, a dirty floating point value might be cast to an integer for some fancy exponent manipulation via bitwise integer logical operations... we want to sanitize it before it goes there. So we want to implicitly sanitize sloppy results that target the integer register file.

- In summary, in fast mode, the hardware only checks for exceptions
 - a. when the result of the operation is targeted at the store register file (sloppy operation -> SRF, implicit sanitizing) or

- b. when the result of the operation is an integer (implicit sanitizing) or
- c. when the operation is a SELECT operation and the destination of the operation is the store register file. (neutral -> SRF, implicitly sanitizing) or
- d. when the operation is a MV_CHECK (explicit sanitizing).

5.3.2 NO TRAP MODE

The final mode that the user can operate in is fully optimized "I don't CARE about exceptions, just give me the answer fast" mode. The compiler can schedule operations without regard to floating exceptions, and the OS disables all exception traps when the process starts. This is the mode that the system currently operates in.

CHAPTER 6

MULTIPLY ACCUMULATE

6.1 DESCRIPTION

The basic MAC operation multiplies its two operands and either adds (macadd) its product to or subtracts (macsub) its product from an accumulated sum or difference. Because the accumulated value is kept inside the floating point functional unit, it not necessary to read it from the register file, and therefore we can perform flops at a higher rate than normal.

The macinit operation is used to initialize the value in the accumulator - it takes its two operands, adds them together, and preserves the sum.

This mac implementation has these characteristics:

- A mac operation must always follow another mac, or macinit on the same functional unit.
- A mac or macinit operation that is not followed by a mac operation will drain the accumulator to the specified destination. A mac or macinit that is followed by a mac will not drain the accumulator.

The mac operations drain to allow the accumulator to be context switched - on a trap, any macs in progress drain to their specified destination. As trap code resumes the process, it must execute a macinit operation to restart the mac pipeline in the same instruction as it TRTNs.

The exception to the single drain register rule is the last MAC in a sequence: this MAC can be allowed to drain anywhere it pleases, since it doesn't need to be restarted and would drain anyway.

6.1.1 MAC OPERATIONS

6.1.1.1 Mac initialize

Options	Resource Class	Notes
(None)	RUC_DP_MACINIT	

- `macinit.f64 0xc4`
Add `src1+src2`, and loads the sum into the accumulator. If not followed by a mac, the accumulator drains to specified destination, which may not be a non-local FRF write (ie, destinations `fb0`, `fb1`, `fb2`, and `fb3` are illegal; destinations `fb`, `lsb`, `lib` are correct).

6.1.1.2 Mac

Options	Resource Class	Notes
(None)	RUC_DP_MAC	
<ul style="list-style-type: none">• macadd.f64 0xc6 Accumulator = Accumulator + src1 * src2. drain to dest if not followed by another mac. Must always follow a macinit or a mac. It may not target fb0, fb1, fb2 or fb3.• macsub.f64 0xc7 Accumulator = Accumulator - src1 * src2. drain to dest if not followed by another mac. It may not target fb0, fb1, fb2, or fb3.		

MicroUnity Lifts Veil on MediaProcessor

New Architecture Designed for Broadband Communications

by Michael Slater



Startup companies based on original microprocessor architectures don't come along very often. Even rarer is an architecture that breaks new ground in instruction-set design, or an implementation that is so much faster than previous chips that it makes possible entirely new applications. And startups that build their own fabs are virtually unheard of. Yet MicroUnity Systems Engineering, a secretive and much-watched startup, has laid out its plans to do all this and more.

MicroUnity, which now has nearly 200 employees, has been the subject of frequent speculation since its founding in 1989 by MIPS cofounder and former IBM researcher John Moussouris. At the recent Microprocessor Forum, Moussouris finally described the MediaProcessor architecture and provided an overview of its first implementation—the result of more than 500 man-years of effort. When fabricated in the company's proprietary BiCMOS process, the MediaProcessor is expected to run at clock frequencies up to 1 GHz—three times greater than the fastest microprocessor today.

The company's funding partners—which have backed the company with more than \$100 million so far—still have not been identified (see sidebar "The People Behind MicroUnity" at end of article). Indeed, MicroUnity's device appears to be at least a year away from volume production, with years more before it can reach its potential. The company already has an income stream from technology licensing but expects to require additional equity investments to help fund the production ramp.

This investment is justified, MicroUnity hopes, by the large future market for communications systems. MicroUnity's goal is that MediaProcessors will someday be ubiquitous throughout communications networks, serving as the universal processing engine both at central "head-end" facilities and in end-user equipment.

The First Broadband Microprocessor

All of MicroUnity's efforts stem from two central premises: that broadband digital communications will be widespread, and that a very fast, yet general-purpose, microprocessor will be the central building block.

The MediaProcessor is intended to deliver not just a few times the performance of today's high-end microprocessors, but hundreds of times their performance—at least on the targeted signal-processing applications. But this is much more than just another supercharged DSP: the MediaProcessor is a general-purpose device that also has exceptional signal-processing capabilities.

This performance level is necessary for the MediaProcessor to handle the data rates needed to implement broadband modulation and demodulation. By implementing techniques like quadrature amplitude modulation (QAM) in software, the device is able to exist in a wide range of different environments and deal adaptively with changing signal and media conditions.

The MediaProcessor can also implement the multimedia functions targeted by conventional multimedia processors, including video compression and decompression, audio processing, and 3D rendering. Compute-intensive DSP tasks such as echo cancellation can be handled as well, along with packet protocols for communications links, cryptography, and routing. It can also run general-purpose applications and operating systems; a 64-bit version of OSF Unix has been ported. This enables the chip to serve as the CPU in head-end servers.

MicroUnity believes that by combining all these functions in a single, programmable processor, the cost of systems ranging from cable modems and interactive set-top boxes to cellular base stations and head-end switch servers can be significantly reduced. Instead of requiring an assortment of special-purpose ASICs, DSPs, and microcontrollers, each with its own private memory, all functions are performed by a single device.

The result is a dramatic reduction in chip count, and potentially in cost. Equally important is the system's agility; unlike a typical ASIC implementation, in which specific algorithms are hard-wired in the chip designs, everything in a MediaProcessor-based system is "soft." In a world where standards are still rapidly evolving, this could be a significant advantage; ASIC-based solutions will become more integrated, but it will be hard for them to match the agility of the MediaProcessor. The MediaProcessor aims to bring to communication devices the same software-based approach that micro-



John Moussouris, CEO of MicroUnity, describes the unusual design of the MediaProcessor.

MICHAEL MISTACH

Storage and Synchronization
load (unsigned) 8/16/32/64/128 bits little/big-end (aligned) (immediate)
store 8/16/32/64/128 bits little/big-end (aligned) (immediate)
add/compare/multiplex (and swap) 64-bit (immediate)
Branch
branch and-equal/and-not-equal/less/less-equal zero
branch equal/not-equal/less/greater-equal
branch floating-point equal/not-equal/less/greater-equal 16/32/64/128
branch (immediate) (and link)
branch gateway (immediate)
branch down/back
Fixed-Point
Data types: 64-bit scalar
group: 128x1/64x2/32x4/16x8/8x16/4x32/2x64/1x128
add/sub (immediate) (overflow)
(unsigned) multiply (and add)
(unsigned) divide
and/or/and-not/or-not/xor/xnor/nor/nand (immediate)
multiplex
(unsigned) set/sub equal/not-equal/less/greater-equal (immediate)
(unsigned) shift/rotate right/left (immediate) (overflow)
(unsigned) compress/expand/extract (immediate)
select bytes
shuffle/deal/swizzle
(unsigned/merge) deposit/withdraw immediate
(shuffle) 4/8-way multiplex
and sum of bits
log most significant bit
Galois-field multiply/polynomial multiply-divide 8/64-bits
Floating-Point
Data types: half/single/double/quad precision
group: 6x16/4x32/2x64
add/sub/mul/div (near/truncate/floor/ceiling/exact)
abs/neg/sqr/sink/float/inflate/deflate (near/truncate/floor/ceiling/exact)
set equal/not-equal/less/greater-equal

Table 1. The MediaProcessor instruction set is unusual in that most instructions can function in group as well as scalar modes. "(*)" indicates optional features; "/" separates alternatives.

processors brought to control systems two decades ago.

The agility may have significant advantages in adapting to varying communications environments. Cable modems, for example, have proved to be very difficult to get working reliably and have to be tuned to each cable system; a MediaProcessor-based cable modem could adapt to the communications channel.

Architecture Blends RISC and DSP

The MediaProcessor instruction-set architecture is a 64/128-bit RISC-style design with several unique features. At the heart of the architecture is the assumption

that software will operate on data items of a wide range of sizes. Unlike conventional architectures, which waste much of the available memory and register bandwidth when using operands of less than the maximum width, the MediaProcessor architecture aims to make full use of the bandwidth, regardless of the data size, by supporting parallel subfield operations. Similar operations are being added to other architectures for multimedia support, but no other CPU is likely to offer as extensive a set of functions or data types as the MediaProcessor.

While some of the operations performed by MediaProcessor instructions are more complex than those in conventional RISC designs, the overall structure is simplified even further. The register file consists of 64 registers of 64 bits each, which can also be accessed as 32 register pairs of 128 bits each. Unlike conventional architectures, there is no separate floating-point register file; all data types use the single unified file.

Furthermore, there are no special registers as part of the user state; the register file and the 64-bit program counter are the entire user state. Condition codes are stored in general registers. Other controls that usually are implemented in special registers, such as floating-point rounding or byte-ordering selectors, are included in the opcodes. These characteristics are thus specified on an instruction-by-instruction basis, eliminating them from the machine state. This simplification of the user state facilitates deeply pipelined designs and the implementation of precise exceptions.

Regular Instruction Encodings

All instructions are encoded in one of a few 32-bit formats. The basic format is an 8-bit opcode and four 6-bit operand fields. For instructions requiring fewer operands, one or more of the fields can be used for immediate data or for opcode extensions. The four-operand format is unusual and allows more complex instructions to be implemented in a single cycle. One price is an additional read port on the register file.

Table 1 summarizes the instruction set. Many of the instructions are typical of any processor, though they have some unique twists. Other instructions are quite different from those in any conventional processor.

Fixed-point data types of 1, 2, 4, 8, 16, 32, 64, and 128 bits are supported, along with four floating-point resolutions: IEEE-standard single- and double-precision, plus 16-bit "half-precision" and 128-bit quad-precision formats. Loads and stores for all data types are provided, each with little- or big-endian data. Unaligned accesses are allowed. Addressing modes are base register plus a 12-bit offset or base register plus index register.

A variety of atomic synchronization instructions support coordination of tasks within one processor or across multiple processors. These include add and swap, compare and swap, and multiplex and swap.

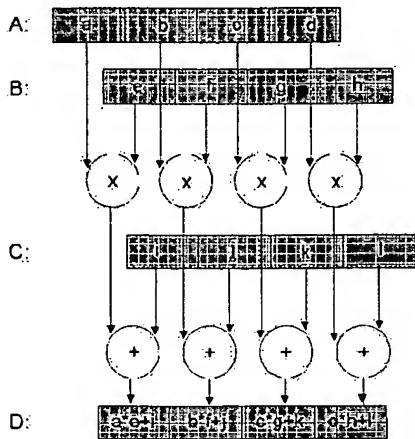


Figure 1. Group floating-point multiply-add operating on four sets of single-precision floating-point numbers.

Branches are not delayed. Both fixed- and floating-point compare-and-branch instructions are provided. The special "gateway branch" changes the privilege level and also loads two 64-bit values from memory: one to reset the program counter to effect the branch, and another to load into a register for use as a data pointer.

In addition to all the usual arithmetic and logical operations, several more unusual operations are provided. These include:

- Find most/least significant "1" in word.
- Count each "1" in word (population count).
- Multiplex. On a bitwise basis, operand register A selects whether the bit from register B or register C is written to the destination register.

Several special arithmetic instructions support extended math functions for signal processing. Only one of these instructions has been disclosed; it implements Galois field arithmetic, which is useful for techniques such as Reed-Solomon error correction.

The architecture supports 64-bit virtual addresses with arbitrary page sizes. It provides four privilege levels in the TLB and 16-bit address-space identifiers as part of the virtual address. All addresses are 64 bits.

Group Ops Support DSP Algorithms

The signal-processing orientation of the instruction set is most evident in the group operations, which are an optional form of most fixed- and floating-point instructions. These instructions are similar to the multimedia extensions added to some RISCs, such as in Sun's UltraSparc, in that they operate on the full register width but subdivide it into multiple data items.

The MediaProcessor's group instructions go beyond those implemented in other processors in that they support floating-point as well as integer instructions and a wider range of data sizes and functions. These instructions operate on 128-bit register pairs divided into fixed-

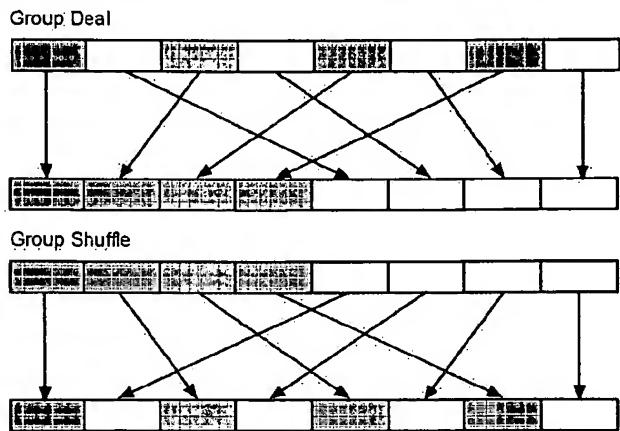


Figure 2. Group deal and group shuffle instructions, shown here operating on 16-bit data.

point sizes of 1, 2, 4, 8, 16, 32, or 64 bits, or 16-, 32-, or 64-bit floating-point values. For example, a single instruction could operate on 32 four-bit values, 16 eight-bit values, or two double-precision floating-point values.

Group operations include multiply, add/subtract, and shift/rotate. Figure 1 shows the most complex group operation: multiply and add. This is a four-operand instruction, operating on 384 bits of input data and delivering 128 bits of result. The figure shows this operation on single-precision floating-point data, allowing four sets of calculations to be performed in parallel. For the integer version of this instruction, only 64-bits of registers A and B are used, so the result can carry twice the number of bits as the input data.

Another set of group instructions, called switching operations, performs complex rearrangements of the data within a register pair. Figure 2 shows two examples: group deal and group shuffle. The switching logic in the processor is able to perform any random rearrangement of the bits in a word, but it takes several instructions to provide enough data to specify an arbitrary remapping. Several common switch operations have been encoded in single instructions:

- Compress, which extracts any contiguous half-size field from each subfield (e.g., 16 bits from each of four 32-bit words) and combines them in a 64-bit word.
- Extract, which does the same as compress but starts with 256 bits of data and delivers 128.
- Expand, which does the opposite of compress.
- Swizzle (copy-swap), which reverses the order of the bits in each subfield within a register pair.

Extensive Software Support

With a new architecture, the burden of software support is great, and MicroUnity has invested heavily in this area as well as in the silicon itself. A number of simulators have been used to support the creation and

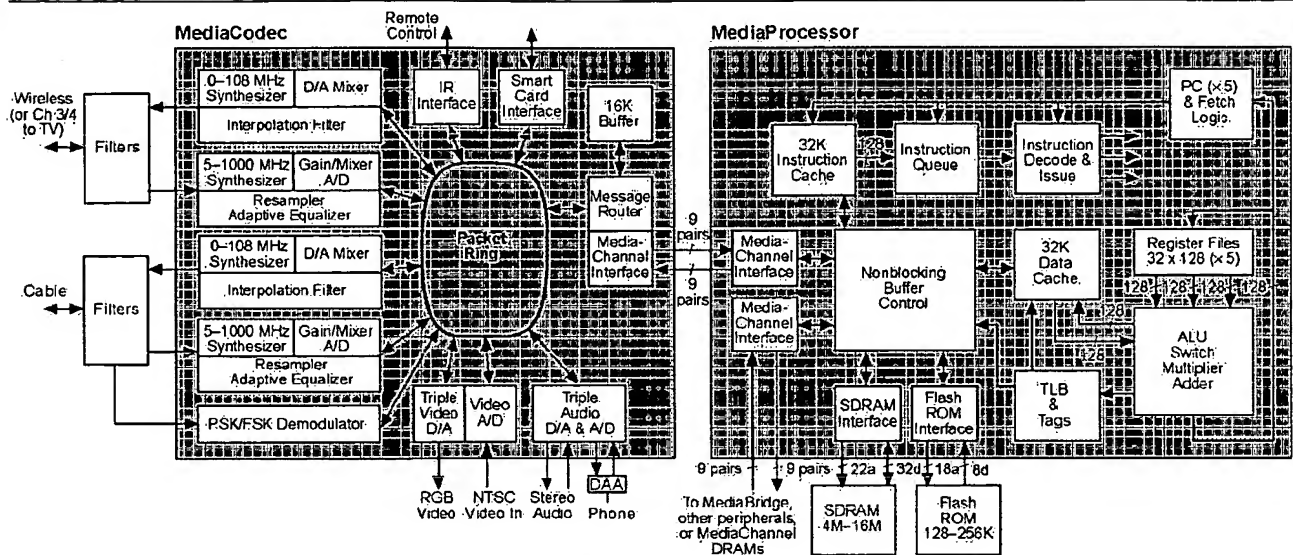


Figure 3. The MediaProcessor provides a direct connection to SDRAM and flash ROM, while using the MediaChannel to connect to the MediaCodec and MediaBridge (not shown). The MediaCodec implements two agile digital radios and other peripheral interfaces.

debugging of a wide range of software, even though no hardware is yet running.

The architecture is supported by C and C++ compilers, as well as a range of simulators, a full 64-bit Unix implementation (for servers), and a real-time kernel (for client devices). Libraries include algorithms for audio (MPEG, Dolby AC-3), video (MPEG-1, MPEG-2, and NTSC encode and decode), communications (QAM and QPSK modems, Reed-Solomon error correction, DES encryption, and broadband MAC protocols), and network protocols (Ethernet, MPEG control, and ISDN).

Chip Set Implements MediaComputer

MicroUnity has completed the design of its first MediaProcessor and two support chips. Figure 3 shows the block diagram for a MediaComputer using the processor and the MediaCodec. A second support chip, not shown here, provides a bridge from the MediaChannel interface to conventional DRAM and PCI bus devices.

The design of all three chips has been completed for many months, but initial test wafers turned up difficulties with delamination of thin insulator strips between wide metal traces in the company's unique process (see "Why Build a Fab" sidebar below). Design-rule changes to remedy the problems have been made, and a revised layout of the processor is now in fabrication. The first functional samples are expected by year-end. Revised layouts for the MediaBridge and MediaCodec will be created after the process and design-rule refinements are proved using the MediaProcessor.

Two versions of the MediaProcessor will be built. One, designed to run at up to a 1-GHz clock rate, will be built in MicroUnity's fab using the company's propri-

etary BiCMOS process. This device was designed to meet the performance needs of a full-function broadband MediaComputer, of which a digital set-top box is a subset. A second design, logically identical but implemented in a 0.5-micron CMOS process and built by an outside foundry, has a target clock rate of 300 MHz. This chip is designed to meet the performance needs of an agile, broadband digital radio, such as a high-speed cable modem or a cellular base station. Tape-out of the CMOS version is scheduled for this month.

MicroUnity's BiCMOS process provides exceptional density. With 10 million transistors on a 100-mm² die (see Figure 4), the MediaProcessor is about one-third the die size of other leading-edge processors that have fewer transistors and are built in 0.5-micron CMOS processes. The tight 1.0-micron metal pitch on the first four interconnect layers is responsible for the density advantage.

The BiCMOS version is projected to dissipate a toasty 85 W at 1 GHz. While this is far more than any commercial microprocessor, MicroUnity claims that this is less than the total power consumed by all the devices it can replace in applications such as a high-quality videoconferencing system. To be sure, it focuses this heat in a smaller area; an impressive heat sink and good air-flow are required. For less demanding applications, the clock rate can be reduced; at 200 MHz, power dissipation is a more modest 14 W.

The CMOS version is expected to dissipate 40 W at 300 MHz. (Both the CMOS and BiCMOS designs use a 3.3-V power supply.) The high power consumption is due, in part, to the fact that the logic design and micro-architecture were designed for bipolar circuits and ported directly to CMOS. Three-fourths of the power is

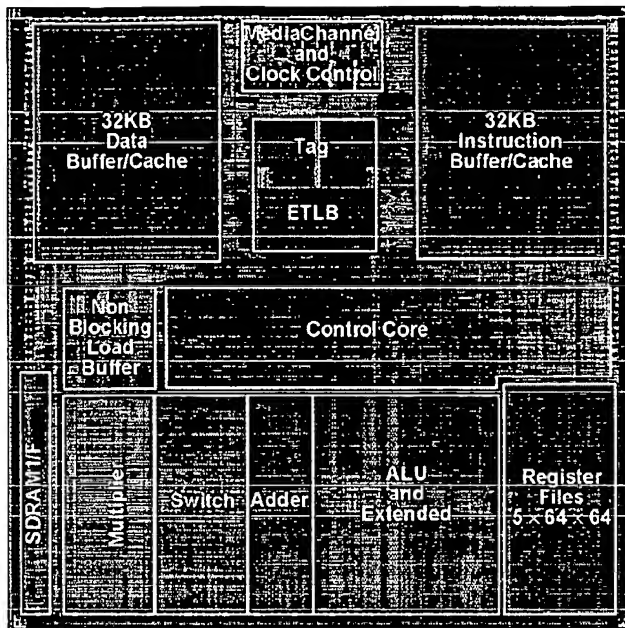


Figure 4. Die plot for the MediaProcessor chip, which integrates 10 million transistors on a die that measures 10 × 10 mm using MicroUnity's 0.5-micron five-layer-metal BiCMOS process.

dissipated in the clock drivers; by putting these drivers off chip, MicroUnity expects to cut power dissipation to 10 W. Future versions are planned to reduce this power consumption even further through a combination of more power-optimized designs and lower supply voltages. The CMOS design has a few more transistors, for a total of 10.5 million, and the die size is 290 mm² using a three-layer-metal process with a 2.0-micron metal pitch.

Each of the other chips uses the same 100-mm² die size in the BiCMOS process. This size is one-fourth of the reticle used in MicroUnity's steppers, which is a sweet spot in production capacity. All three chips are packaged in identical 312-lead TAB packages. The processor has 197 signal leads and 115 power and ground connections. MicroUnity uses a silicon interposer between the chip and the TAB frame to spread the leads and provide thermal-expansion matching.

Superpipelined and Supertthreaded

Designing a microprocessor to operate at 1 GHz presents some formidable challenges, and the resulting microarchitecture is quite different from today's processors. In addition to a very deep pipeline, MicroUnity's proprietary process, low-voltage-swing on-chip signals, and special circuit and logic design techniques all play a role in achieving the high clock rate. The design is not superscalar, eliminating the complexity of multiple-instruction dispatch, dependency checking, out-of-order execution, and other techniques that are nearly universal in today's high-end microprocessors. Like conventional processors, the MediaProcessor has a pipelined

Why Build a Fab?

For a startup to build its own fab is almost unheard of today. Why did MicroUnity do this? The company wanted to build high-performance, very high speed mixed-signal chips, and the founders believed that no existing process met this need—and they had some innovative ideas that would create a faster, denser process than others had created. The company has already been granted more than two dozen patents, covering the process as well as the architecture and implementation techniques, and many more are pending.

The crunch in foundry availability also played a role. The company's first implementation was designed for a foundry, but the owner of that fab decided to get out of the foundry business. Other founders were asking for up-front payments of tens of millions of dollars to provide access to capacity. MicroUnity built a small fab for not much more, and in the process created some valuable intellectual property that it can license to others.

MicroUnity developed a 0.5-micron BiCMOS process with very high density. The effective gate length is 0.3 microns. MicroUnity has designed its process and its fab to scale to 0.35-micron drawn dimensions.

The process uses five metal layers. The first four all have a tight 1.0-micron metal pitch, which provides greater connection density than any production process today. (Intel comes close, but others are further behind.)

Instead of conventional aluminum interconnect, MicroUnity uses a gold alloy. Another exotic feature is that metal layers 3 and 4 can optionally be air-bridged; the insulating material below the trace is etched away, leaving the conductor floating in air. This reduces capacitance, power consumption, and crosstalk noise.

The process is extremely planar (flat within 0.1 micron at every layer). In the metal layers, this is done with proprietary lift-off techniques; unlike conventional processes, no polishing is used because current polishing technology doesn't work with gold interconnect.

The CMOS structures are used for memory cells, while the bipolar circuits are used for logic and analog functions. Logic circuits use a MicroUnity-developed constant-current variable-bias ECL design.

The fab has 9,000 ft² of clean-room space and is designed to produce 5,000 6" wafers per month when fully equipped. Though the fab is small by today's standards, MicroUnity believes that its process's high density will give it greater chip volume than larger fabs. So far, the company has spent less than \$50 million to build the fab and install the equipment to deliver about one-third of its full capacity. When the fab is fully outfitted, the company expects the total cost to be about \$100 million.

MicroUnity's chip designs use a sea-of-gates ASIC design approach with automatic routing and timing optimization, minimizing design time and making the designs easily adaptable. Custom macros have been created for memories and analog blocks.

Price & Availability

Development systems based on the MediaProcessor are expected to be available in the first half of 1996, along with samples of the processor chip in both CMOS and BiCMOS versions. Volume production is planned for later 1996. Samples of the MediaBridge and MediaCodec are expected in the first half of 1996, with production in early 1997. Pricing has not yet been set.

For further information, contact Tony Steliga, VP Product Marketing, MicroUnity Systems Engineering (Sunnyvale, Calif.) at 408/734-8100; fax 408/734-8141; e-mail tony@microunity.com.

execution unit, instruction and data caches of 32K each, and a paged memory-management unit.

A look inside the execution unit, as Figure 5 illustrates, reveals a level of pipelining far deeper than any other microprocessor. Just the execution and data access pipeline—not counting instruction fetch, register read, and write-back—is 15 stages deep. Each pipeline stage consists of a single (but up to 16×16 wide) and-or-invert gate, followed by a latch.

The logic blocks within the pipeline are similar to those in conventional processors, though the ALU is more complex, to handle the group and extended math (e.g., Galois field) instructions. The ALU takes three 128-bit inputs and produces a 128-bit result. There is an additional switch block that rearranges data for the group switching instructions. The multiplier is followed by an adder to support multiply-add instructions.

The initial MediaProcessor does not implement the

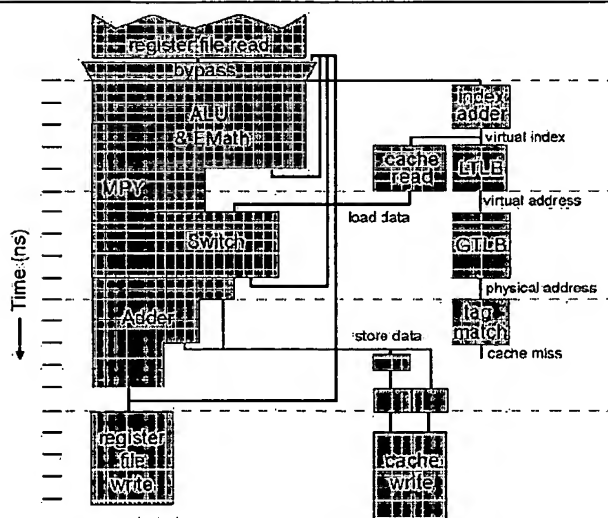


Figure 5. The MediaProcessor is superpipelined to support the high clock rate. Each tick on the left shows one pipeline stage, or 1 ns at the 1-GHz clock rate. The dashed lines every five stages show the major cycle, which is the clock rate seen by each of the five threads.

floating-point aspects of the architecture. The company took the 100-mm² die size as a limit; within this constraint, implementing the FPU would have meant a substantial reduction in the cache size. A floating-point version of the processor has been designed, and future implementations will include FP capability. The FP instructions will simplify some signal-processing algorithms and improve their performance, because they eliminate the need for scaling.

Supertreading Reduces Latency

Design at the 1-GHz target frequency has some special challenges. Moving a signal through a wire just a third of the way across the chip takes close to one clock cycle (1 ns), so some pipeline stages have nothing in them but a wire and a latch.

In a conventional design, such a deep pipeline would result in long latencies, sapping performance. MicroUnity overcame this problem by designing the chip to support five concurrent execution threads, a technique they call supertreading. There are five copies of the register file and program counter. Since there are no special registers, this provides five complete sets of user state. Each set is accessed on successive clock cycles; during a five-clock period, each set has one clock cycle of access to the pipeline, caches, memory, and I/O system.

In effect, the processor runs five separate threads in parallel, with each thread having the performance of a 200-MHz processor. The major benefit of this approach is that the effective latency (as seen by an individual thread) of the arithmetic units, register file, and memory system is divided by five. Only a single load-delay slot—which can frequently be hidden by the compiler—is needed, just as in many conventional RISCs.

One cycle of the 1-GHz clock is called a minor cycle, while five cycles—one for each thread—is called a major cycle. In terms of the latency seen by each thread, only the major cycles count. Simple ALU operations have a one-cycle latency; switch and load operations have a two-cycle latency; and multiply and multiply-add operations have a three-cycle latency.

The register file has three read ports and one write port. It is accessed in a single clock, supporting the maximum bandwidth of four operands per instruction.

The processor implements only a simple branch prediction algorithm. Five branch target buffers store the target address of the most recent backward branch for each thread, so after the first pass through a loop instruction, fetching can follow the loop without delay. The penalty for a mispredicted or unpredicted branch is three cycles. One reason the MediaProcessor architects decided not to devote silicon area to full branch prediction is that the instruction set includes special features, such as the multiplex instruction, that eliminate many branches.

Large Caches Backed by SDRAM

The on-chip instruction and data caches, 32K each, are each able to deliver 128 bits every two clock cycles. As a result, the instruction cache actually has twice the bandwidth required to keep the pipeline fed. During a 10-cycle interval (two major cycles), each thread gets one double-word access to each cache. The caches are direct-mapped, virtually indexed, and virtually tagged.

The on-chip caches can also be used as a software-managed buffer memory. All, three-fourths, or one-half of each cache can be allocated as buffer memory, with the remainder operating as cache.

The MediaProcessor includes a 32-bit-wide SDRAM interface that supports up to 16M of memory and a dedicated byte-wide interface for up to 512K of flash memory to store firmware. A nonblocking buffer in the memory interface allows up to 16 pending load or store operations, and threads are not blocked until they need the result of a load operation or the buffer is full.

Careful design of the threads minimizes contention for the memory. For example, in a set-top box application, one thread handles radio demodulation and a second handles audio processing; the data sets for these threads fit in the on-chip memory. MPEG-2 decoding is divided among three threads, which coordinate their processing so their memory accesses do not conflict.

Systems needing more than 16M of DRAM must use a MediaBridge chip, which provides an interface to up to 128M of DRAM, as well as a PCI bus for other peripherals. Each MediaBridge also includes a 128K buffer that can be software-controlled or can operate as a second-level cache. Up to four MediaBridge chips can be connected to a single MediaProcessor.

MediaChannel Provides 1-Gbyte/s Path

To provide a low-cost yet high-speed connection to peripheral chips, MicroUnity took an approach similar to that used by Rambus: a very fast but only 8-bit-wide connection called the MediaChannel.

As Figure 6 shows, the MediaChannel supports one master and up to four slave devices. By using a point-to-point, unidirectional connection, a simple protocol, and special timing circuits, very high clock rates are achieved: the MediaChannel runs at the same clock rate as the processor (1 GHz for the BiCMOS implementation). Despite this high clock rate, relatively long traces—and even high-quality ribbon cable—can be used because of the point-to-point design and the differential, low-voltage-swing (100 to 700 mV) signals.

The clock is carried as a ninth signal, using identical circuits to the data. Programmable skew calibrators in each device, which can adjust the delay with a resolution of tens of picoseconds, compensate for variations in delays of the chip and connecting wires. This adjustment

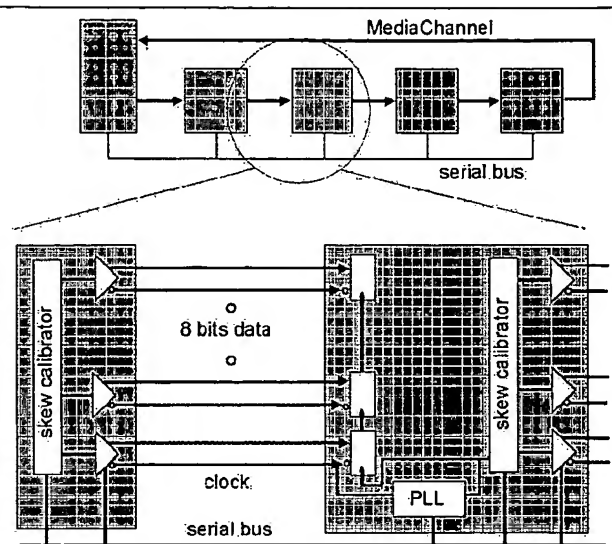


Figure 6. The MediaChannel interconnect uses a ring with one master and up to four slave devices. With a 1-GHz clock rate, it delivers a peak bandwidth of 1 Gbyte/s.

process and other diagnostics are handled by a simple, low-speed serial bus that connects all the devices.

All MediaChannel packets are fixed length, and the protocol is streamlined to support high data rates. Read transactions begin with a six-byte request packet, which includes a header byte, a four-byte address, and a check byte. The header byte identifies the type of packet as a read or write request (with or without allocate, for slave devices with caches), read or write response, idle, or error. A read response packet consists of a header byte, eight bytes of data, and a check byte.

Write transactions are initiated by a packet containing a four-byte address and eight bytes of data, framed by header and check bytes. A write response is just a header and check byte.

The MediaProcessor provides two separate MediaChannel interfaces. Either can be used for one or more MediaCodec or MediaBridge chips. MicroUnity is working with DRAM manufacturers to develop MediaChannel DRAMs, allowing the channel to be used directly for memory expansion.

MediaCodec Implements Digital Radios

Although the MediaProcessor provides the computational speed needed to process broadband signals, it is the MediaCodec that brings these signals into the digital domain. With about 10 million transistors, the MediaCodec rivals the complexity of the MediaProcessor. As a very high speed mixed-signal design, in which noise and other problems can wreak havoc, it is the most challenging of the chips to bring from concept to production.

As Figure 3 shows, the MediaCodec includes two bidirectional RF interfaces. In a set-top box, one would

The People Behind MicroUnity

John Moussouris, chairman and CEO of MicroUnity, was a cofounder and VP of R&D at MIPS Computer Systems. Before MIPS, he was a founding member of an IBM team that developed a VLSI implementation of the landmark "801" RISC processor.

The company's chief technologist, and the man behind its unique process technology, is Al Matthews. A veteran of many high-end process designs, he has worked on bipolar, CMOS, and GaAs process development at Performance Semiconductor, Avantek, Gould, Hewlett-Packard, and Intel.

The chief architect of the MediaProcessor is Craig Hansen, who managed the architecture of the R2000 and R3000 at MIPS. He also developed floating-point architectures for Weitek and Hewlett-Packard.

An important force behind the scenes is William Randolph Hearst III, former publisher of the *San Francisco Examiner* and currently a partner at Kleiner Perkins Caufield & Byers, as well as CEO of @Home, Inc.

Other board members include Lois Abraham, a senior partner at Brown & Bain and MicroUnity's general counsel; John L. Doyle, retired executive VP of Hewlett-Packard; and Bruce Ravenel, one of the original 8086 architects and now senior VP and COO of TCI Technology Ventures.

HP and TCI (the dominant operator of cable television systems in the U.S.) are both reportedly major investors in the company, as is Will Hearst. Other investors reportedly include Microsoft and Motorola. Motorola is rumored to be interested in the process technology, as well as the applications to its communications business.

connect to the cable, and the other would drive a wireless cell or an RF-modulated signal to the television. This chip is a superset of all the capabilities requested by all of MicroUnity's partners, and any individual application could use a simpler version if the company were to develop such versions. MicroUnity is designing a CMOS MediaCodec that uses external analog circuitry for the highest-speed functions; this chip will be more cost-effective for applications like cable modems.

The two RF interfaces are each capable of tuning a channel of up to 8 MHz from a center frequency of 5 to 1,000 MHz. Combined with processing in the MediaProcessor, a digital bandwidth of up to 40 Mbps can be extracted from each channel (using schemes such as 64-256QAM).

An on-chip packet ring connects the modules within the MediaCodec. The chip appears to the MediaChannel as a 16K block of memory, plus additional address space for memory-mapped control registers. The on-chip message router moves data between the dual-ported memory and the I/O channels.

A Long Road to High Volume

Although the initial MediaProcessor chip set will have some commercial applications, it is most significant as a proof of concept and as a development vehicle. Development systems, and even some applications, will be able to get by without the MediaCodec, which won't be available as soon as the processor.

The initial implementation is designed to deliver all the capabilities needed for a range of applications; future designs promise to offer higher integration and much lower power. The company is now working on a super-scalar MediaProcessor designed to be implemented in CMOS. The superthreaded approach of the initial design was optimized for BiCMOS technology; for a CMOS implementation, a superscalar design running at a lower clock rate will be more power efficient.

Another factor in reducing the power consumption of future designs is lower supply voltages. MicroUnity projects that MediaProcessors eventually will be built with supply voltages approaching 1 V, with a power consumption of under 1 W—a level that must be reached for portable and low-cost consumer devices. The company also plans eventually to integrate the MediaProcessor and MediaCodec on a single chip, along with additional memory. This would result in a single-chip device that could take in an RF signal, demodulate and process the signal, and provide a digital or RF-modulated output.

In the near term, MicroUnity has more immediate challenges building and debugging the initial chip set. Not only are there all the risks associated with establishing a new architecture, but there are also the challenges of bringing up a new fab using a process technology radically different than any implemented elsewhere. When technology is pushed in so many directions, there are multiple opportunities for things to go wrong.

The company is not entirely dependent on its process technology, however; the chips can be built with foundry processes, albeit with lower performance levels. The company is also licensing its architecture and chip designs. Should the markets for broadband MediaComputers develop too slowly, the company can apply its technology to existing applications, such as cellular base stations.

Skeptics point out that it is extremely rare for any company to get as far ahead in clock rate as MicroUnity claims to be. Until working MediaProcessors are demonstrated, some skepticism is appropriate. If MicroUnity is able to meet its technology goals, its challenge then will be to find a role in the marketplace. That there will be massive growth in the communications infrastructure can be taken for granted, but the anticipated shape of the new structure seems to evolve weekly. In this regard, MicroUnity's agility, open licensing strategy, and corporate partnerships (which have yet to be announced) could prove to be the critical factors. ♦

Motion Video Instruction Extensions for Alpha

Paul Rubinfeld
Bob Rose
Michael McCallig

Contact:
Paul Rubinfeld
Semiconductor Engineering Group
77 Reed Road, HLO2-3/D11
Hudson, MA 01749

prubinfeld@ds.hlo.dec.com

18-Oct-96

Scope

The Alpha architecture[1] has added several new instructions called Motion Video Instructions (*MVI*) that can be used to accelerate the performance of key algorithms used in emerging motion video technologies. The criteria for selecting these new instructions are based on the fundamental premise that it is important to keep the Alpha architecture "clean" in order to facilitate extremely fast circuit implementations of the architecture. In order to be added, a new instructions must pass two critical tests: the benefits provided must span multiple generations; and, the benefits must be material.

The MVIs for the Alpha architecture were motivated by the desire to perform high quality software motion video encoding using the prevalent ISO/ITU video compression standards. The standards that were addressed were MPEG 1 [2], MPEG 2 (also known as H.262) [3], H.261 [4], and H.263. These are all motion compensated discrete cosine transforms (*DCT*) and inverse discrete cosine transforms (*IDCT*) based compression methods. Thus, a common set of hardware features are able to accelerate the computations for each of these standards.

MPEG 1 is targeted at storing 74 minutes of CD quality audio and VHS quality video on a single speed CD ROM. MPEG 2 is a scaleable compression standard that is applicable to low resolution video through High Definition Television. The most prevalent application is broadcast quality television at approximately 6Mb/sec. H.261 is aimed primarily at ISDN teleconferencing and is scaleable in resolution and quality from a single BRI (128 Kb) up to a full T1 line. H.263 is a new teleconferencing standard that should eventually replace H.261. It is scaleable from sending video and audio over a 28.8 kbaud analog modem to very high bit rates and high resolution.

In all cases, the standard published by ISO or the ITU is an interchange standard. It specifies what the bit-stream should look like. The implementation of the decoder is pretty much determined by the bit-stream specification. The algorithms used to encode the bit-stream, however, provide for a rich set of tradeoffs affecting cost, image quality, and support of extended features. An encoder design provides value added by producing the highest quality pictures for a for a given bit rate within the constraints of the compression engine. The design of the compression engine involves another set of cost verses quality tradeoffs. Thus, each encoder design adds value by optimizing all of these parameters to produce the best video at the lowest cost for a given design point.

Software encoders are available today for H.261 and H.263 at low resolution and low frame rate. Software real time MPEG encoders are not commonly available today. Most of the software encoders today use highly simplified motion estimation algorithms. This is necessary because motion estimation is an extremely large consumer of CPU cycles. In order to meet real time constraints, it is necessary to make a number of serious compromises. This results in video that looks good when there is little motion. However, when there is a lot of complex motion, the quality falls off rapidly. The quality of the motion estimation is, in most cases, the largest single factor limiting the quality of the video in an implementation.

The goal of the new Alpha MVIs is to enable software encoders that produce quality competitive with dedicated hardware encoders. Thus, the motion estimation techniques that are described below are the same techniques used by many hardware encoders.

The motion estimation operation is far and away the highest consumer of CPU cycles. Typically, the motion estimation workload is an order of magnitude higher than the workload other video operations.

New Instructions

The new instructions are:

1. **PERR Ra.rq, Rb.rq Rc.wq** - Sum of Absolute Differences - The absolute value of the differences between each of the bytes in Ra and Rb is calculated. The sum of these resulting bytes is written to Rc. This instruction provides an order of magnitude performance improvement over code written without the MVI instructions and enables useful software encoding of Motion Compensated Video.
2. **MINUB8, MINSB8, MINUW4, MINSW4, MAXUB8, MAXSB8, MAXUW4, MAXSW4** - Clamping Instructions - These permit minimum and maximum operations on multiple 1-byte (*byte*) or 2-byte data (*word*) stored in an 8-byte data word (*quadword*). Both signed and unsigned versions are supported. At various places in the standard video algorithms, it is necessary to limit the range of results by saturating them to a value. This has been done with a CMOV instruction in the past. These new instructions provide better performance than CMOV because of their 8-way parallelism.
3. **UNPKBL, UNPKBW, PKLB, PKWB** - Packing and unpacking bytes in quadword from words, and 4-byte data words (*longwords*). In motion video, these are useful instructions because pixels are stored as bytes and words, the DCT and IDCT must be done as longwords, and the motion compensation needs 9-bits. The algorithms need to convert the IDCT results from longwords to words, the reference image data from bytes to words, add the two values, then clamp the result and convert the final results back to bytes. Similar examples for using these instructions can be made for generic signal processing examples that use 8-, 16-, or 32-bit data.

Analysis of Motion Estimation Computations

The following analysis assumes that the reader is familiar with the various ISO standards. We will look at several motion estimation algorithms that are frequently used in commercial-quality hardware MPEG encoders (full search estimation, hierarchical search estimation and telescoping search estimation)

Full Search Estimation

The purpose of Motion Estimation is to create a predicted frame from a reference frame or frames already available to the decoder. For each macroblock, a search of the reference frame

is conducted in its vicinity. The predicted frame is created by tiling together the best match found by motion estimation for each macroblock. The encoder then sends the instructions to create the predicted frame and the difference between the predicted frame and the actual frame.

The distance searched for each macroblock is not explicitly set by the ISO standards. Typically, a moving object will cover a similar distance between each frame. Therefore, the practical *search range* needed depends on the number of frames between the reference frame and the current frame. The larger the frame spacing, the larger the search range needs to be. For example, in our MPEG 1 encoder, we search ± 22.5 pixels for frames that are three frames apart.

The criterion for best match is also not explicitly set the ISO standards. The most common match algorithm uses the macroblock position that produces the minimum sum of absolute differences. The sum of absolute differences is calculated by subtracting each pixel in the macroblock in the reference frame from each pixel in the macroblock in the new frame. The following equation is evaluated for each search position

$$E(dx, dy) = \sum_{i=0}^{15} \sum_{j=0}^{15} |r(x_0 + dx + i, y_0 + dy + j) - p(x_0 + i, y_0 + j)|$$

where: x_0 = the X position of the macroblock
 y_0 = the Y position of the macroblock
 dx = x offset from macroblock X position
 dy = y offset from macroblock Y position
 r is the addressed pixel from the reference frame
 p is the addressed pixel from the new frame

The value of dx and dy that produced the lowest error value form x_0 and y_0 coordinates is the motion vector. The number of computations for one search is 256 differences, 256 absolute values, and 255 sum operations.

Hierarchical Motion Estimation

The number of computations can be reduced relative to the full search technique by doing a hierarchical search. In this method, the reference image and the new image are subsampled to half the vertical and half the horizontal resolution. The subsampled images are use to search over the full search area to get the high order bits of the motion vector. Then, the search is done at full resolution over a narrow range around the vector resulting from the subsampled search. Finally, the search is done on the half pixel positions around the full pixel motion vector. For example, if a 2:1 subsample factor is used for the first search, the number of computations to perform one search is 64 differences, 64 absolute value operations, and 63 sums. This is reduction from full pixel search by a factor of 4. In addition, one search covers four pixel positions, so one fourth the number of searches are required. This represents a factor of 16 improvement. However, it is still necessary to conduct the full pixel search over a narrow range and a half pixel search over a ± 0.5 pixel area. Taking this into account reduces the improvement to a factor of about 10 instead of 16.

There maybe a small quality loss with the hierarchical search algorithm compared to the full search algorithm of the same search range. However, since the hierarchical algorithm permits a larger area to be search in a fixed amount of time, it may actual improve quality.

Telescoping Search Estimation

The distance required for the search can be reduced by using a telescoping search. In a sequence of frames (I, P, B are specified in the ISO standard, the subscript numbering denote frame sequence):

$I_1 B_2 B_3 P_4$

a typical set of search ranges would be:

Search	Range
$I_1 \rightarrow P_4$	± 22.5
$I_1 \rightarrow B_2$	± 7.5
$P_1 \rightarrow B_2$	± 15.5
$I_1 \rightarrow B_3$	± 15.5
$P_4 \rightarrow B_3$	± 7.5

The search range is increased as the distance between frames increases to allow for the fact the objects in motion had more time to move. In telescoping search, the searching is done in frame sequence order. The trajectory for a given macroblock is used to predict the new position of the macroblock. That prediction is use to initialize the search for the macroblock in the next frame. This permits searching over a smaller range of pixels. Reverse searches are done in reverse frame order using a similar learning initialization method.

For the above case, $I_1 \rightarrow B_2$ is conducted first using the ± 7.5 pixel search range. Next, the $I_1 \rightarrow B_3$ search is performed centered around a prediction from the position found for the $I_1 \rightarrow B_2$ search. Assuming the motion is smooth from one frame to the next, the search range for $I_1 \rightarrow B_3$ can be reduced to ± 7.5 pixels. Next the $I_1 \rightarrow P_4$ search is conducted with a ± 7.5 pixel range starting from the position found from the $I_1 \rightarrow B_3$ search. Next, the $P_4 \rightarrow B_3$ search is conducted followed by the $P_4 \rightarrow B_2$ search. The bottom line here is that all of the search ranges are reduced to the range used for an adjacent frame.

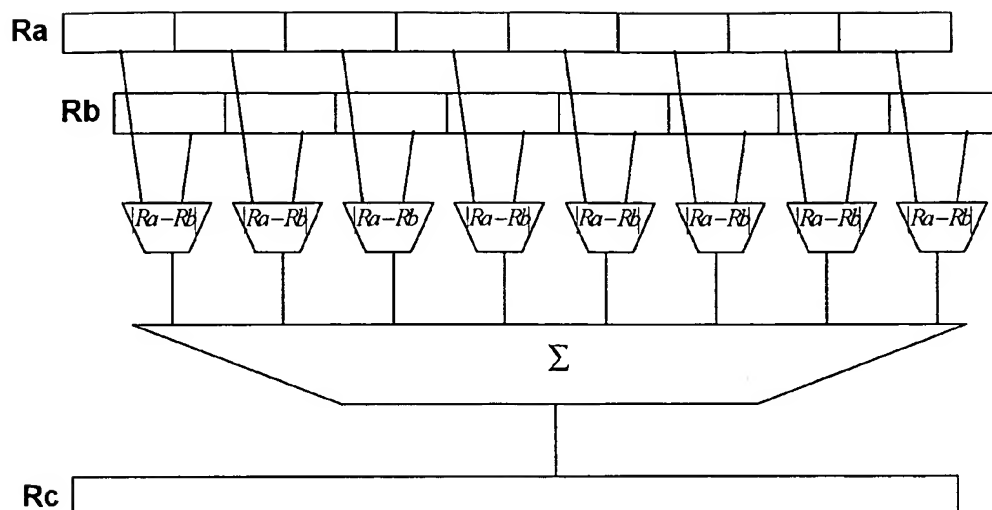
The telescoping search works well for smooth motion. It can fail to find the optimum motion vector for erratic motion in some cases. It works well in typical video sequences.

Selected Method

The above methods are used today in high-quality hardware MPEG encoders. The high-performance Alpha architecture with the new MVI instruction will allow these algorithms to be done in software in real-time.

PERR Instruction

Each search method uses the PERR instruction. It takes eight pixels packed into quadword and computes the absolute differences between two registers. This is shown in the figure on the top of the next page:



PERR retires eight pixels of motion compensation computation. The net result is that eight pixels of motion estimation work can be accomplished in one clock tick of Alpha.

Analysis

The cycle counts will be computed for several interesting ways of doing motion estimation. These number will then be used to compute the cycle counts for doing motion estimation for different frame sizes and frame rates.

A given macroblock search is done in three steps: first, a 2x2 subsample search is used to cover the full search area. This produces a motion vector good to the best 2 pixel position. Next, the $\pm 1 \times \pm 1$ area around this motion vector is searched at full resolution to find the best motion vector to 1 pixel position. Finally, the half pixel positions around this motion vector are searched to find the best half pixel motion vector. The new Alpha processors with the MVI instructions are fast enough to support this search strategy for a real time encoder.

2x2 Subsampled Search

The operands require eight registers each, so both macroblocks fit in registers in the Alpha architecture. The eight pixel wide operations cover the horizontal span of the subsampled macroblock. The overhead of loading the registers is minimized by conducting the search in a vertical direction so that for the search inner loop, the top scan is discarded and the bottom scan is added. This permits the X alignment work to be saved from one macroblock to the next.

As noted in the previous section, Alpha can do the absolute differences, tree summation, accumulation for one scan to the next, and loads for the next search all in one clock tick. The loop control can be done in two more clock ticks. The total cycle count allocated for one search position is 10 clock ticks.

Full Search

Full search technically means searching every half pixel position. The cycle count for this is prohibitive. A good approximation is to search every full pixel position then find the best half pixel position in the neighborhood of the full pixel match. That is the case that will be analyzed here.

A 16X16 pixel macroblock does not fit in the Alpha integer registers. The impact is minor for a hierarchical motion estimator because the full pixel search range is small.

Each search position requires summing the absolute difference of each pixel over 16X16 pixels. The new instructions can search eight pixels in one clock tick. The search is conducted vertically so the X alignment work can be saved. However, it must be stored and loaded since it does not fit in registers. Each eight pixels of work require two loads, a sum of absolute differences, a tree add and an accumulate. The remainder of the overhead can be buried in the remaining cycles available to the Alpha execution units. So, a full pixel search requires 64 CPU cycles.

Half Pixel Search

The half pixel search requires the half pixel data to be interpolated by averaging the macroblock with a macroblock offset by one pixel. Once the interpolated reference data is created, the match operation is conducted in the same matter as the full pixel search. The interpolation consists of eight cases. Four cases; up, down, left and right, require averaging of two positions. The four diagonal cases require averaging of four positions.

Each scan requires 3 *load* instructions and two *align instructions* to create the reference data. The offset data can be created with two more align instructions for X offsets and with one *mov reg* instruction for Y offsets. It should be possible to do all the averaging work and pixel differencing work in 3 ticks for each eight pixels. That results in 96 cycles for each position searched.

Cycle Counts

The following chart shows the important parameters of several frame formats of interest. The cycle count is computed for the various resolutions assuming that a hierarchical search is conducted starting with a 2X2 subsampled search to cover the bulk of the search range. This is followed by a ± 1 pixel search at full resolution and a $\pm 1/2$ pixel search. The computation is done with and without using telescoping search. The data shows that the telescoping search cuts the work load about in half.

For CCIR601 resolution, the computation was done for an MPEG 1 type of motion estimation using the same search range as SIF. This should produce results close to what you would get for MPEG 2 field coding.

	I Frames	P Frames	B Frames
# per second	2	8	20

# of Searches	Dist 1	Dist 2	Dist 3	Cycles/Search
Subsampled	49	225	529	10
Full Pixel	9	9	9	64
Half Pixel	9	9	9	96

Cycles/Mblock				
Subsampled	490	2,250	5,290	
Full Pixel	576	576	576	
Half Pixel	864	864	864	
Total	1,930	3,690	6,730	

	QSIF	SIF	CIF	CCIR601
Width	176	352	352	720
Height	120	240	288	480
# Pixels	21,120	84,480	101,376	345,600
Mblocks/Frame	88	330	396	1,350
Frame Rate	30	30	30	30

Hierarchical Srch				
Dist 1 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 2 cycles/sec	6,494,400	24,354,000	29,224,800	99,630,000
Dist 3 cycles/sec	4,737,920	17,767,200	21,320,640	72,684,000
Total	14,629,120	54,859,200	65,831,040	224,424,000

Telescoping Srch				
Dist 1 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 2 cycles/sec	3,396,800	12,738,000	15,285,600	52,110,000
Dist 3 cycles/sec	1,358,720	5,095,200	6,114,240	20,844,000
Total	8,152,320	30,571,200	36,685,440	125,064,000

Conclusions

The added instructions provide the ability to do motion estimation for QSIF, SIF and CIF resolutions using 10% - 15% on Alpha. CCIR601 resolution uses about half the Alpha. These usage figures can be cut approximately in half if a telescoping search is employed. A telescoping search should produce high quality results for well behaved motion sequences. More random sequences may show some quality loss. The cycle count analysis needs to be extended to cover all of the motion estimation modes in MPEG 2.

Marginal Benefit of Adding Intel-like Multimedia Instruction to Alpha

Several vendors have recently added multimedia instructions (MMX instruction set) to the X86 Architecture [5]. These instructions feature 16-bit integers packed into a 64-bit operand so that one instruction can do four 16-bit operations. The cornerstone on the new instructions is the MUL-ADD instruction, which does four 16-bit multiplies and accumulates the 32-bit products into two 32-bit sums. Interestingly, Intel did not include any instructions that accelerate the pixel error

calculation similar to PERR. This precludes Intel from ever providing a meaningful (near-real time) software motion video estimation solution that meets the ISO/IEC standards.

This section examines why Alpha did not further extend the architecture to include other MMX-like instruction.

The basic reasons the Alpha instruction set was not further expanded to include the new Intel-like extensions (most notably the MUL-ADD instruction) are that 16-bit data size is insufficient for a lot of important applications including the commercial motion video market; memory bandwidth limitations minimizes the overall performance benefit provided by these new instructions; and, any real performance benefit that Alpha would get from such extensions would be small since MMX essentially tries to fix some of the x86 legacy-architecture performance deficiencies which are not present in the Alpha architecture (e.g., limited number of registers, poor floating point organization, etc.)

Limitations to 16-bit processing

There are extreme limitations to processing that can be done with 16-bit data. Even in the motion video markets which only utilizes a highly constrained set algorithms, more than 16-bits are required to be compliant with the ISO/IEC standard algorithms that are based on motion compensation and DCT transform coding. These include MPEG 1 and MPEG2, H.261 teleconferencing and H.263. Adding 16-bit functions is inconsistent with the goal to create MVI instructions that are intended to support these standards and enable growth in algorithm complexity for these markets.

In addition, scientific uses of signal and image processing always require significantly more than 16-bits of accuracy, especially on intermediate results. A considerable amount of pre-programming analysis and scaling within the code is avoided by using the high dynamic range of floating point. The 16-bit limit of MMX comes into play in two ways. First, to prevent overflow, the dynamic range of the data will be less than 16 bits even with the fact that the MMX multiplier product carries the full 32 bits. A typical value for a maximum 16 bits is to restrict the data to 8-10 bits. Since round off noise averages 6 dB/bit, the noise threshold on processing with MMX-like instructions is 50 - 60 dB. It is not enough for scientific applications. Scientific processing also uses very large data records. This is the second way 16-bit processing is a limiting factor. A 32-point fast Fourier transform (FFT) can be calculated with 16-bit processing. A 128 point FFT can often be calculated with 16-bit processing (it is dependent on the exact values in the data record). A million point FFT can never be calculated with 16-bit processing.

Limitations Due to Memory Bandwidth

The marginal benefit of further extending the Alpha instruction set can be seen to be limited for most commercial applications due to memory bandwidth issues. The examples will be drawn from three classes of codes

1. Small codes, where the data set always hits in the cache and the total cycles is a very small percentage of the available cycles.
2. Medium codes, where the cache performance is very good but there are misses. Total cycles is a noticeable percentage of the available cycles.
3. Large code, where the cache performance is poor and total cycles is a large percentage of the available cycles.

The results will show that only medium size codes might benefit from further adding MMX-like instructions to Alpha. Small codes execute so fast that any speedup is unimportant. For large codes, the most important performance issue is memory access, not CPU cycles. MMX-like instructions can do nothing to help. Medium codes might benefit from MMX like instructions. This is a fairly narrow class of problems since it is really a class that sits on the border between codes that fit in cache and codes that do not. The fact that the problem must be computable with 16-bits makes it smaller still. It is unlikely that the performance improvement for a restricted set of codes is worth the addition of MMX-like instructions. It is also true that many of these codes, such as the integer DCT used as an example, can be attacked in other ways. In the case of the discrete cosine transform, sparse matrix techniques are used.

Small Codes

Smaller problems can be handled with fewer bits, and the non-scientific markets referred to above do have relatively small signal and image processing needs. Small problems fit in on-chip caches. It is unlikely that such code is the performance bottleneck in a larger application.

An example is the 4x4 matrix-vector multiply, commonly used to manipulate 3D objects [6]. One matrix is multiplied with many different 4 element vectors. Reference [6] gives the total instructions, but not the schedule or cycle count.

Intel without MMX [6]	72 instructions
Intel with MMX [6]	28 instructions

The Alpha processor will do the calculation in floating point. The loop is unrolled 2 times. The compiler generated code keeps 10 of the 16 elements of the 4x4 matrix in registers. The remaining 6 are loaded within the loop as are the two 4 element vectors. The floating point operation count for Alpha for the twice unrolled loop is

loads	14
multiplies	32
adds	24
stores	8

There are also loop and address instructions that multiple issue with the floating point operations. The multiple issue breakdown is

single issues	29
dual issues	20
triple issues	2
quad issues	2
stall cycles	0

total cycles 53 for 2 matrix - vector multiplies

We assume perfect pairing for the MMX code and a 200MHz Intel processor. The Alpha clock is 500 MHz.

Intel with MMX	14 Intel cycles = 35 Alpha cycles
Alpha	26 Alpha cycles

If you allow for pairing to not be perfect on the Intel processor, the Alpha is faster by a factor approaching 2. MMX-like instructions could speed up this code on Alpha. Would it be worth it? AI-

pha can do 19 million 26 cycle operations per second. If only 1% of the CPU time were spent on this operation, Alpha could do 190,000 matrix-vector multiplies per second. This is far more than needed in a typical multimedia application. Alpha can satisfy the need for matrix-vector multiplies and spend less than 1% of the CPU time doing so. Clearly, for this example, the improvements of adding MMX-like instructions to MVI do not pass the "significant benefits" test outlined at the beginning of this paper.

The alert reader will suspect that even 1% is overstated. The premise of this example, and indeed of all the MMX examples in the Intel application notes, is that the data is in the on-chip cache. The Intel processors have a 16KB on-chip data cache and Alpha has up to 96KB on-chip. To stay in these caches, the number of matrix-vector multiplies must be on the order of 8000 (typically it is much higher). Alpha can do this in less than 5 hundredths of one percent of the total processor time ($26 * 8000$ cycles out of 500,000,000). If there is a processor bottleneck, this is not it.

Large problems

Larger multimedia problems are memory or even disk bound. Adding MMX-like instructions to Alpha would not lead to a speedup. An example is an image dissolve [6]. One image fades out while another fades in over it. The equation the MMX example is based on is

$$\text{result_pixel} = \text{image_1} * (k/255) + \text{image_2} * (1 - k/255)$$

where k is a constant within each frame and there are 8 bits in each pixel color. As k is varied from 255 to zero, image_1 fades out and image_2 fades in. The best MMX performance was obtained when the image was organized as separate color planes. The MMX instructions could then be used to advantage by processing 4 pixels from a single plane at once. The example from reference [6] is, unlike the matrix-vector multiply example, a real performance issue. The question is, what is limiting the performance and will MMX-like instructions help Alpha? The example images are 640 x 480 pixels (full screen standard TV), 8 bits per pixel and 3 colors. The total data for the two images is

$$2 * 640 * 480 * 1 * 3 = 1.8 \text{ Megabytes}$$

The example does the fade over 255 frames, or about 8.5 seconds. The total image data processed is 470 Megabytes. The performance bottleneck here is the memory system, not the CPU. The addition of MMX-like instructions will not help.

Medium problems

"Using MMX Instructions in a Fast IDCT Algorithm for MPEG Decoding" [7] describes and gives a hand crafted code listing for an integer discrete cosine transform using MMX instructions. The IDCT is intended for use in an MPEG decoder for displaying compressed video. The MPEG compression/decompression algorithm uses the IDCT on an 8x8 array of 16 bit data. The code provided in [7] does the IDCT in 240 Intel cycles. The skillfully hand-crafted code pairs instructions in 200 of the 240 cycles.

It should be noted in passing that the IDCT algorithm used is not compliant with the IEEE specification for IDCT accuracy and would lead to unacceptable artifacts in a video conferencing application using the H.261, H.262, and H.263 specifications. It is tolerable for home applications of MPEG decoding.

To evaluate the need for MMX-like instructions on Alpha for MPEG decoding, the MPEG decoder from the MPEG Software Simulation Group [8] was compiled, run and profiled. A 150

frame sequence was decompressed into 352x240 color frames. The file of compressed video is 725 Kbytes. The IDCT in this code is IEEE compliant which is required for commercial applications.

The functions that used more than half a percent of the CPU time are shown in figure 1. To focus on processor performance, time reading compressed data or writing decompressed data are not included.

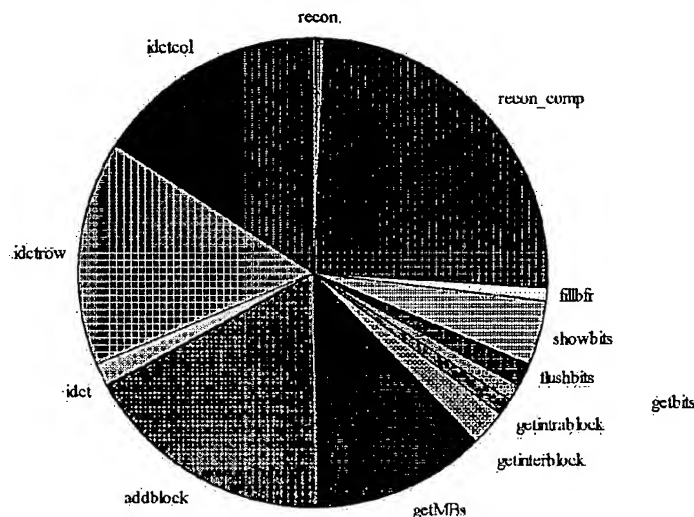


Figure 1. MPEG Decoder Execution Profile

The first point to notice is that there are several functions besides the IDCT that take a sizable amount of the processor.

IDCT (idct idctrow idctcol)	33 %
recon_comp	25 %
addblock	17 %
get_MBs	12 %

87 % of the time is spent in these four areas. Only the 33 % in the IDCT can be helped by MMX instructions. If we get a 2 times speedup, that would translate into 15% on the MPEG decoder.

The IDCT code was written to take advantage of all zero valued coefficients in any row or column of the 8x8 matrix being operated on. The 8x8 IDCT is done by doing eight 8x1 IDCT's on the rows followed by eight 8x1 IDCT's on the columns. The 8 coefficients are loaded and tested for all zero. If they are zero, then a quick exit can be done without the need for the full calculation. The full calculation takes 3 times longer than the quick exit. On Alpha, the cycle count for a complete 8x8 IDCT is 656 cycles.

There is no way to do the quick exit with the MMX code [7] without significantly increasing the schedule. The MMX code has had to unroll the IDCT to such an extent that branch decisions such as this can no longer be made.

Intel with MMX
Alpha

240 Intel cycles = 600 Alpha cycles
656 Alpha cycles

IDCT Scheduled Cycles

Scheduled cycles is only part of the story. Since the data being processed is larger than the on-chip cache, the actual number of cycles will be greater than the scheduled cycles, the difference being the time to access memory and/or the board cache. Reference [7] does not provide any execution times. On Alpha, the off-chip accesses take an additional 325 cycles. If it is assumed that the Intel off-chip accesses are in the same proportion, then an estimate of IDCT time for both processors can be calculated. In the table below, both processors are assumed to have the same absolute time to off-chip cache or memory. The Alpha clock is 2.5 times the Intel clock. All values are in Alpha cycles.

	Alpha	Intel
total scheduled cycles	650	600
off-chip cycles	325	325
total	975	925

IDCT Actual Cycles

An obvious next step is to try to eliminate the off-chip cycles by prefetching the data during arithmetic operations. This requires that the processor continue to execute instructions that do not depend on the data being fetched. Both Alpha and the Intel Pentium Pro can do this. This will get the IDCT down to 20% of the MPEG decoder. MMX-like instructions could then cut this in half, saving 10% on the decoder. It is not clear that this savings is worth the addition of a new class of instructions.

Conclusion

MMX-like instructions have been examined on Alpha in the context of three examples from Intel application notes on MMX. The other examples from these notes have been studied and all fall into one of the three categories described here:

1. The code and data are small enough that it is not a performance bottleneck
2. The code and data are so large that memory, rather than CPU performance, is the bottleneck.
3. The code and data are in an intermediate range where MMX-like instructions could boost performance on the order of 10%.

As with most performance issues, there are several related factors that are difficult if not impossible to break out individually. The most widely discussed advantage of MMX for X86 processors is the four fold parallelism. Another equally important advantage is a four fold increase in the apparent size of the register file. This in turn allows loop unrolling, which allows instruction scheduling to avoid unused issue slots. It is equally valid to consider this expansion of the register set as the first addition to the architecture, with most of the performance advantage deriving from it. The MMX instructions are then just a very clever way of implementing the expanded register set without the hardware difficulties of actually doing so. Alpha has the large register file, has the unrolling, and has a compiler to automatically schedule the code. When looked at from this perspective, it is understandable that MMX would not be nearly as significant for Alpha processors as it is for Intel processors.

It is interesting that Intel did not include a PERR instruction in the MMX extensions. This instruction provides the most computational benefit of all architectural extensions. PERR enables real time or near-real time software video encoding solutions, something that is not possible with Intel's MMX solution.

Further extending Alpha to include MMX-like instructions would not pass the critical tests outlined at the beginning of this paper. The benefits that would be provided would not span multiple generations (the application which can utilize 16-bit accuracy are already very limited) and the actual benefits added by such extensions are very minimal.

References

- [1] Alpha Architecture Reference Manual, R. Sites, ed., Digital Press, Burlington, Ma, 1992
- [2] ISO/IEC 11172 MPEG-1 Standard, Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5 Mbits/sec
- [3] ISO/IEC MPEG-2 Standard
- [4] ITU Standard H.261, Video Codex for Audiovisual Services at P x 64 Kbits/sec
- [5] Press release MMX technology, Intel, March 5, 1996.
- [6] MMX Technology Overview, Intel.
- [7] Using MMX Instructions in a Fast iDCT Algorithm for MPEG Decoding, Intel.
- [8] MPEG-2 Encoder / Decoder, Version 1.1, June 1994, MPEG Software Simulation Group

An Architectural Overview of the Programmable Multimedia Processor, TM-1

Selliah Rathnam, Gert Slavenburg

Philips Semiconductors
811 E. Arques Avenue, Sunnyvale, CA 94088

ABSTRACT

TM-1 is the first in a family of programmable multimedia processor from the Trimedia product group of Philips Semiconductors. This "C" programmable processor has a high performance VLIW-CPU core with video and audio peripheral units designed to support the popular multimedia applications. TM-1 is designed to concurrently process video, audio, graphics, and communication data. The VLIW-CPU core is capable of executing a maximum of twenty seven operations per cycle, and the sustained execution rate is about five operations per cycle for the tuned applications. The audio unit easily handles different audio formats including the 16-bit stereo data. The video unit is capable of processing different YUV and RGB pixel formats with horizontal and vertical scaling and color space conversion. TM-1 applications

can range from low-cost, stand alone systems such as video phones to programmable, multipurpose plug-in cards for traditional computers.

1.0 INTRODUCTION

TM-1 is a building-block for high-performance multimedia applications that deal with high-quality video and audio. TM-1 easily implements popular multimedia standards such as MPEG-1 and MPEG-2, but its orientation around a powerful general-purpose CPU makes it capable of implementing a variety of multimedia algorithms, whether open or proprietary.

More than just an integrated microprocessor with unusual peripherals, the TM-1 microprocessor is a fluid

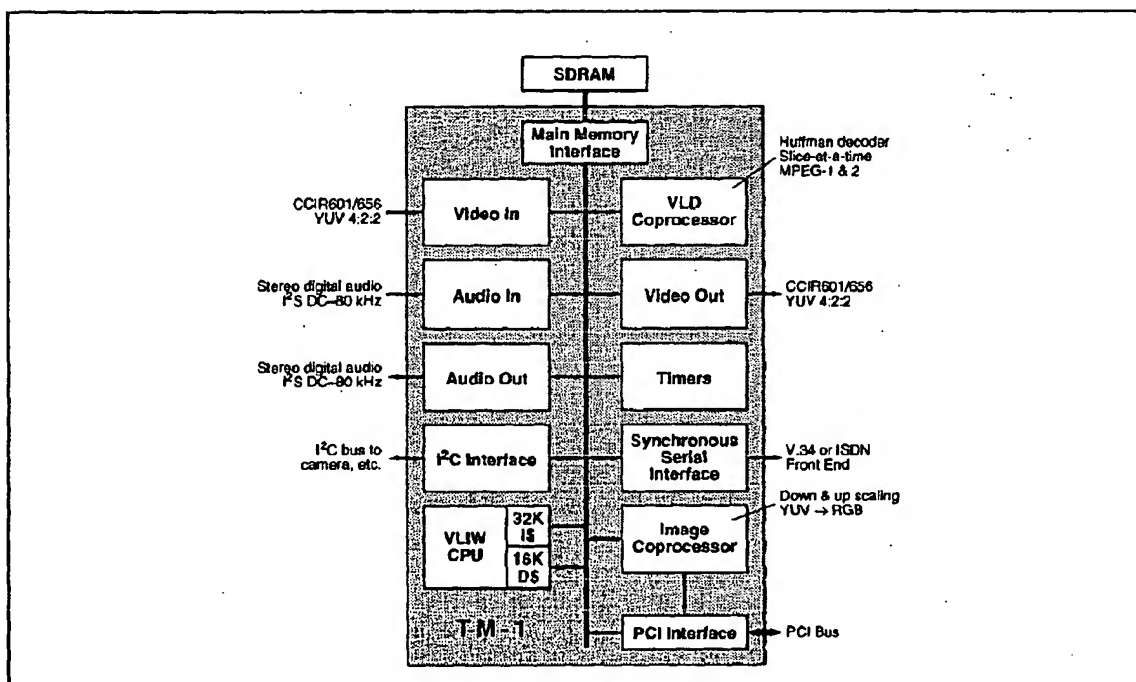


Figure 1. TM-1 block diagram.

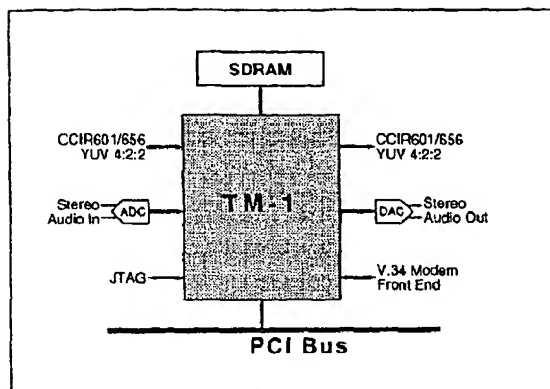


Figure 2. TM-1 system connections. A minimal TM-1 system requires few supporting components.

computer system controlled by a small real-time OS kernel that runs on the VLIW processor core. TM-1 contains a CPU, a high-bandwidth internal bus, and internal bus-mastering DMA peripherals.

TM-1 is the first member of a family of chips that will carry investments in software forward in time. Compatibility between family members is at the source-code level; binary compatibility between family members is not guaranteed. All family members, however, will be able to perform the most important multimedia functions, such as running MPEG-2 software.

Defining software compatibility at the source-code level gives Philips the freedom to strike the optimum balance between cost and performance for all the chips in the TM-1 family. Powerful compilers ensure that programmers seldomly need to resort to non-portable assembler programming. Programmers use TM-1's powerful low-level operations from source code; these DSP-like operations are invoked with a familiar function-call syntax. Trimedia also provides hand-coded and tuned multimedia libraries which can be used to increase the performance of the multimedia applications.

As the first member of the family, TM-1 is tailored for use in PC-based applications. Because it is based on a general-purpose CPU, TM-1 can serve as a multi-function PC enhancement vehicle. Typically, a PC must deal with multi-standard video and audio streams, and users desire both decompression and compression, if possible. While the CPU chips used in PCs are becoming capable of low-resolution real-time video decompression, high-quality video decompression—not to mention compression—is still out of reach. Further, users demand that their systems provide live video and audio without sacrificing the responsiveness of the system.

TM-1 enhances a PC system to provide real-time multimedia, and it does so with the advantages of a special-purpose, embedded solution—low cost and chip count—and the advantages of a general-purpose processor—reprogrammability. For PC applications, TM-1 far surpasses the capabilities of fixed-function multimedia chips.

Other Trimedia family members will have different sets of interfaces appropriate for their intended use. For example, a TM-1 chip for a cable-TV decoder box would eliminate the video-in interface.

2.0 TM-1 CHIP OVERVIEW

The key features of TM-1 are:

- A very powerful, general-purpose VLIW processor core that coordinates all on-chip activities. In addition to implementing the non-trivial parts of multimedia algorithms, this processor runs a small real-time operating system that is driven by interrupts from the other units.
- DMA-driven multimedia input/output units that operate independently and that properly format data to make processing efficient.
- DMA-driven multimedia coprocessors that operate independently and perform operations specific to important multimedia algorithms.
- A high-performance bus and memory system that provides communication between TM-1's processing units.

Figure 1 shows a block diagram of the TM-1 chip. The bulk of a TM-1 system consists of the TM-1 microprocessor itself, a block of synchronous DRAM (SDRAM), and minimal external circuitry to interface to the incoming and/or outgoing multimedia data streams. TM-1 can gluelessly interface to the standard PCI bus for personal-computer-based applications; thus, TM-1 can be placed directly on the PC mainboard or on a plug-in card.

Figure 2 shows a possible TM-1 system application. A video-input stream, if present, might come directly from a CCIR 601-compliant digital video camera chip in YUV 4:2:2 format; the interface is glueless in this case. A non-standard camera chip can be connected via a video decoder chip (such as the Philips SAA7111). A CCIR 601 output video stream is provided directly from the TM-1 to drive a dedicated video monitor. Stereo audio input and output require external ADC and DAC support. The operation of the video and audio interface units is highly customizable through programmable parameters.

The glueless PCI interface allows the TM-1 to display video via a host PC's video card and to play audio via a host PC's sound hardware. The Image Coprocessor provides display support for live video in an arbitrary number of arbitrarily overlapped windows.

Finally, the V.34 interface requires only an external modem front-end chip and phone line interface to provide remote communication support. The modem can be used to connect TM-1-based systems for video phone or video conferencing applications, or it can be used for general-purpose data communication in PC systems.

3.0 BRIEF EXAMPLES OF OPERATION

The key to understanding TM-1 operation is observing that the CPU and peripherals are time-shared and that communication between units is through SDRAM mem-

ory. The CPU switches from one task to the next; first it decompresses a video frame, then it decompresses a slice of the audio stream, then back to video, etc. As necessary, the CPU issues commands to the peripheral units to orchestrate their operation.

The TM-1 CPU can enlist the ICP and video-in units to help with some of the straightforward, tedious tasks associated with video processing. The function of these units is programmable. For example, some video streams are—or need to be—scaled horizontally, so these units can handle the most common cases of horizontal down- and up-scaling without intervention from the TM-1 CPU.

3.1 Video Decompression in a PC

A typical mode of operation for a TM-1 system is to serve as a video-decompression engine on a PCI card in a PC. In this case, the PC doesn't know the TM-1 has a powerful, general-purpose CPU; rather, the PC just treats the hardware on the PCI card as a "black-box" engine.

Video decompression begins when the PC operating system hands the TM-1 a pointer to compressed video data in the PC's memory (the details of the communication protocol are typically handled by a software driver installed in the PC's operating system).

The TM-1 CPU fetches data from the compressed video stream via the PCI bus, decompresses frames from the video stream, and places them into local SDRAM. Decompression may be aided by the VLD (variable-length decoder) unit, which implements Huffman decoding and is controlled by the TM-1 CPU.

When a frame is ready for display, the TM-1 CPU gives the ICP (image coprocessor) a display command. The ICP then autonomously fetches the decompressed frame data from SDRAM and transfers it over the PCI bus to the frame buffer in the PC's video display card (or the frame buffer in PC system memory if the PC uses a UMA (Unified Memory Architecture) frame buffer). The ICP accommodates arbitrary window size, position, and overlaps.

3.2 Video Compression

Another typical application for TM-1 is in video compression. In this case, uncompressed video is usually supplied directly to the TM-1 system via the video-in unit. A camera chip connected directly to the video-in unit supplies YUV data in eight-bit, 4:2:2 format. The video-in unit takes care of sampling the data from the camera chip and demultiplexing the raw video to SDRAM in three separate areas, one each for Y, U, and V.

When a complete video frame has been read from the camera chip by the video-in unit, it interrupts the TM-1 CPU. The CPU compresses the video data in software (using a set of powerful data-parallel operations) and writes the compressed data to a separate area of SDRAM.

The compressed video data can now be disposed of in any of several ways. It can be sent to a host system over

the PCI bus for archival on local mass storage, or the host can transfer the compressed video over a network, such as ISDN. The data can also be sent to a remote system using the integrated V.34 interface to create, for example, a video phone or video conferencing system.

Since the powerful, general-purpose TM-1 CPU is available, the compressed data can be encrypted before being transferred for security.

4.0 VLIW CORE AND PERIPHERAL UNITS

4.1 VLIW Processor Core

The heart of TM-1 is its powerful 32-bit CPU core. The CPU implements a 32-bit linear address space and 128, fully general-purpose 32-bit registers. The registers are not separated into banks; any operation can use any register for any operand.

The core uses a VLIW instruction-set architecture and is fully general-purpose. TM-1 uses a VLIW instruction length that allows up to five simultaneous operations to be issued. These operations can target any five of the 27 functional units in the CPU, including integer and floating-point arithmetic units and data-parallel DSP-like units.

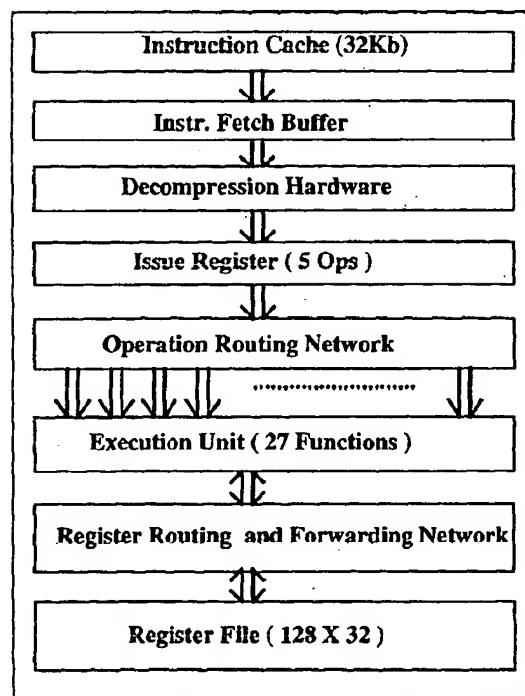


Figure 3. VLIW Processor Core and Instruction Cache.

Although the processor core runs a tiny real-time operating system to coordinate all activities in the TM-1 system, the processor core is not intended for true general-purpose use as the only CPU in a computer system. For example, the processor core does not implement virtual memory address translation, an essential feature in a general-purpose computer system.

TM-1 uses a VLIW architecture to maximize processor throughput at the lowest possible cost. VLIW architectures have performance exceeding that of superscalar general-purpose CPUs without the extreme complexity of a superscalar implementation. The hardware saved by eliminating superscalar logic reduces cost and allows the integration of multimedia-specific features that enhance the power of the processor core.

The TM-1 operation set includes all traditional microprocessor operations. In addition, multimedia-specific operations are included that dramatically accelerate standard video compression and decompression algorithms. As just one of the five operations issued in a single TM-1 instruction, a single special or "custom" operation can implement up to 11 traditional microprocessor operations. Multimedia-specific operations combined with the VLIW architecture result in tremendous throughput for multimedia applications.

4.2 Internal "Data Highway" Bus

The internal data bus connects all internal blocks together and provides access to internal control registers (in each on-chip peripheral units), external SDRAM, and the external PCI bus. The internal bus consists of separate 32-bit data and address buses, and transactions on the bus use a block-transfer protocol. Peripherals can be masters or slaves on the bus.

Access to the internal bus is controlled by a central arbiter, which has a request line from each potential bus master. The arbiter is configurable in a number of different modes so that the arbitration algorithm can be tailored for different applications. Peripheral units make requests to the arbiter for bus access, and depending on the arbitration mode, bus bandwidth is allocated to the units in different amounts. Each mode allocates bandwidth differently, but each mode guarantees each unit a minimum bandwidth and maximum service latency. All unused bandwidth is allocated to the TM-1 CPU.

The bus allocation mechanism is one of the features of TM-1 that makes it a true real-time system instead of just a highly integrated microprocessor with unusual peripherals.

4.3 Memory and Cache Units

TM-1's memory hierarchy satisfies the low cost and high bandwidth requirement of multimedia markets. Since multimedia video streams can require relatively large temporary storage, a significant amount of DRAM is required.

TM-1 has a glueless interface with synchronous DRAM (SDRAM) or synchronous graphics RAM

(SGRAM), which provide higher bandwidth than the standard DRAM. As the SDRAM has been supported by major DRAM vendors, the competition among those vendors will keep the SDRAM price in par with that of the standard DRAM. TM-1's DRAM memory size can range from 2Mbytes to 64 Mbytes.

The TM-1 CPU core is supported by separate 16-KB data and 32-KB instruction caches. The data cache is dual-ported in order to allow two simultaneous load/store accesses, and both caches are eight-way set-associative with a 64-byte block size.

4.4 Video-In Unit

The video-in unit interfaces directly to any CCIR 601/656-compliant device that outputs eight-bit parallel, 4:2:2 YUV time-multiplexed data. Such devices include digital camera systems, which can connect gluelessly to TM-1 or through the standard CCIR 656 connector with only the addition of ECL level converters. Non-CCIR-compliant devices can use a digital decoder chip, such as the Philips SAA7111, to interface to TM-1. Older front ends with a 16-bit interface can connect with a small amount of glue logic.

The video-in unit demultiplexes the captured YUV data before writing it into local TM-1 SDRAM. Separate data structures are maintained for Y, U, and V.

The video-in unit can be programmed to perform on-the-fly horizontal resolution subsampling by a factor of two if needed. Many camera systems capture a 640-pixel/line or 720-pixel/line image; with subsampling, direct conversion to a 320-pixel/line or a 360-pixel/line image can be performed with no CPU intervention. Further, if subsampling is required eventually, performing this function during data capture reduces initial storage requirements.

4.5 Video-Out Unit

The video-out unit essentially performs the inverse function of the video-in unit. Video-out generates an eight-bit, multiplexed YUV data stream by gathering bits from the separate Y, U, and V data structures in SDRAM. While generating the multiplexed stream, the video-out unit can also up-scale horizontally by a factor of two to convert from CIF to native CCIR resolution.

Since the video-out unit likely drives a separate video monitor—not the PC's video screen—the PC itself cannot be used to generate the graphics and text of a user interface. To remedy this, the video-out unit can generate graphics overlays in a limited number of configurations.

4.6 Image Coprocessor (ICP)

The image coprocessor (ICP) is used for several purposes to off-load tasks from the TM-1 CPU, such as copying an image from SDRAM to the host's video frame buffer. Although these tasks can be easily performed by the CPU, they are a poor use of the relatively expensive CPU resource. When performed in parallel by the ICP, these tasks are performed efficiently by simple hardware, which allows the CPU to continue with more complex tasks.

The ICP can operate as either a memory-to-memory or a memory-to-PCI coprocessor device.

In memory-to-memory mode, the ICP can perform either horizontal or vertical image filtering and resizing. The ICP implements 32 FIR filters of five adjacent pixel input values. The filter coefficients are fully programmable, and the position of the output pixel in the output raster determines which of the 32 FIR filters is applied to generate that output pixel value. Thus, the output raster is on a 32-times finer grid than the input raster. The filtering is done in either the horizontal or vertical direction but not both. Two applications of the ICP are required to filter and scale in both directions.

In memory-to-PCI mode, the ICP can perform horizontal resizing followed by color-space conversion. For example, assume an $n \times m$ pixel array is to be displayed in a window on the PC video screen while the PC is running a graphical user interface. The first step (if necessary) would use the ICP in memory-to-memory mode to perform a vertical resizing. The second step would use the ICP in memory-to-PCI mode to perform a horizontal resizing (if necessary) and colorspace conversion from YUV to RGB.

While sending the final, resampled and converted pixels over the PCI bus to the video frame buffer, the ICP uses a full, per-pixel occlusion bit mask—accessed in destination coordinates—to determine which pixels are actually stored in the frame buffer for display. Conditioning the transfer with the bit mask allows TM-1 to accommodate an arbitrary arrangement of overlapping windows on the PC video screen.

Figure 3 illustrates a possible display situation and the

data structures in SDRAM that support the ICP's operation. On the left in Figure 3, the PC's video screen has four overlapping windows. Two, Image 1 and Image 2, are being used to display video generated by TM-1.

The right side of Figure 3 shows a conceptual view of SDRAM contents. Two data structures are present, one for Image 1 and the other for Image 2. Figure 3 represents a point in time during which the ICP is displaying Image 2.

When the ICP is displaying an image (i.e., copying it from SDRAM to a frame buffer), it maintains four pointers to the data structures in SDRAM. Three pointers locate the Y, U, and V data arrays, and the fourth locates the per-pixel occlusion bit map. The Y, U, and V arrays are indexed by source coordinates while the occlusion bit map is accessed with screen coordinates.

As the ICP generates pixels for display, it performs horizontal scaling and colorspace conversion. The final RGB pixel value is then copied to the destination address in the screen's frame buffer only if the corresponding bit in the occlusion bit map is a one.

As shown in the conceptual diagram, the occlusion bit map has a pattern of 1s and 0s that corresponds to the shape of the visible area of the destination window in the frame buffer. When the arrangement of windows on the PC screen is changed, modifications to the occlusion bit maps may be necessary.

It is important to note that there is no preset limit on the number and sizes of windows that can be handled by the ICP. The only limit is the available bandwidth. Thus, the ICP can handle a few large windows or many small win-

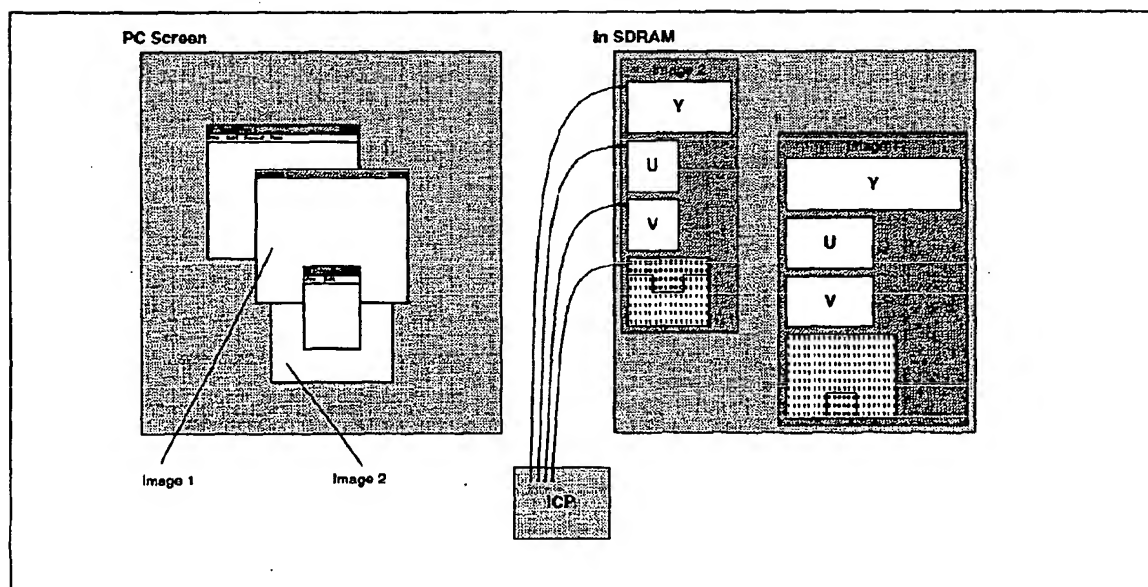


Figure 4. ICP operation. Windows on the PC screen and data structures in SDRAM for two live video windows.

dows. The ICP can sustain a transfer rate of 50 megapixels per second, which is more than enough to saturate PCI when transferring images to video frame buffers.

ICP has a micro-programmable engine. All ICP operations such as filtering, scaling and color space conversions and their formats are programmable. The ICP's micro programs loads itself from the SDRAM memory.

4.7 Variable-Length Decoder (VLD)

The variable-length decoder (VLD) is included to relieve the TM-1 CPU of the task of decoding Huffman-encoded video data streams. It can be used to help decode MPEG-1 and MPEG-2 video streams.

The VLD is a memory-to-memory coprocessor. The TM-1 CPU hands the VLD a pointer to a Huffman-encoded bit stream, and the VLD produces a tokenized bit stream that is very convenient for the TM-1 image decompression software to use. The format of the output token stream is optimized for the MPEG-2 decompression software so that communication between the CPU and VLD is minimized.

As with the other processing-intensive coprocessors, the VLD is included mainly to relieve the CPU of a task that wastes its performance potential. When dealing with the high bit rates of MPEG-2 data streams, too much of the CPU's time is devoted to this task, which prevents its special capabilities from being used.

4.8 Audio-In and Audio-Out Units

The audio-in and audio-out units are similar to the video units. They connect to most serial ADC and DAC chips, and are programmable enough to handle most reasonable protocols. These units can transfer MSB or LSB first and left or right channel first.

The sampling clock is driven by TM-1 and is software programmable within a wide range from DC to 80 kHz with a resolution of 0.02 Hz. The clock circuit allows the programmer subtle control over the sampling frequency so that audio and video synchronization can be achieved in any system configuration. When changing the frequency, the instantaneous phase does not change, which allows frequency manipulation without introducing distortion.

As with the video units, the audio-in and audio-out units buffer incoming and outgoing audio data in SDRAM. The audio-in unit buffers samples in either

eight- or 16-bit format, mono or stereo. The audio-out unit simply transfers sample data from memory to the external DAC; any manipulation of sound data is performed by the TM-1 CPU since this processing will require at most a few percent of the CPU resource.

4.9 PCI Bus Interface Unit (BIU)

This unit connects the internal Data Highway Bus to an external PCI bus. It has a PCI master to initiate memory read/write cycles for TM-1-CPU requested read/write transactions including burst read/write DMA transactions. The PCI target within the BIU responds to the transactions initiated by external PCI master devices to read/write the TM-1's memory space, and it satisfies their requests. External devices can access the TM-1's MMIO registers through this unit.

The ICP unit has a direct connection to the BIU unit in order to transfer the pixel image data efficiently from TM-1 to the graphics device or host memory through the PCI bus.

The DMA transactions are considered as background transactions. To reduce the latency of the single word read/write transactions on the PCI bus, the BIU interleaves the burst read/write DMA cycles with single word read/write transactions.

5.0 CUSTOM OPERATIONS

Custom operations in the TM-1 CPU architecture are specialized, high function operations designed to dramatically improve performance in important multimedia applications. Custom operations enable an application to take advantage of the high performance VLIW-CPU core.

Important multimedia applications, such as the decompression of MPEG video streams, spend significant amounts of execution time dealing with eight-bit data items. Using 32-bit operations to manipulate small data items makes inefficient use of 32-bit execution hardware in the implementation. There are custom operations designed to operate on four eight-bit data items simultaneously in order to improve the performance about four to ten times compared with that of the general purpose CPU. Furthermore, some custom operations are defined to combine multiple arithmetic and control instructions into a single custom operation. These custom operations can be used easily in the C language as function calls.

Custom operation syntax is consistent with the C pro-

```

unsigned char A[16][16];
unsigned char B[16][16];
.
.
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 1)
        cost += abs(A[row][col] - B[row][col]);
}

```

Figure 6. Match-cost loop for MPEG motion estimation.

gramming language, and just as with all other operations generated by the compiler, the scheduler takes care of register allocation, operation packing, and flow analysis.

The multimedia application development has been additionally improved by providing hand coded and well tuned multimedia code in the form of 'C' library functions.

5.1 Example: Motion-Estimation Kernel

One part of the MPEG coding algorithm is motion estimation. The purpose of motion estimation is to reduce the cost of storing a frame of video by expressing the contents of the frame in terms of adjacent frames.

A given frame is reduced to small blocks, and a subsequent frame is represented by specifying how these small blocks change position and appearance; usually, storing the difference information is less expensive than storing a whole block. For example, in a video sequence in which the camera pans across a static scene, some frames can be expressed simply as displaced versions of their predecessor frames. To create a subsequent frame, most blocks are simply displaced relative to the output screen.

The code in this example is for a match-cost calculation, a small kernel of the complete motion-estimation code. This code provides an excellent example of how to transform source code in order to make the best use of TM-1's custom operations.

Figure 5 shows the original source code for the match-cost loop. The code is not a self-contained function. At some location early in the code, the arrays A[] and B[] are declared; At some location between those declarations and the loop of interest, the arrays are filled with data.

We start by noticing that the computation in the loop of Figure 5 involves the absolute value of the difference of two unsigned characters (bytes). TM-1 operation set includes, several operations that process all four bytes in a 32-bit word simultaneously. Since the match-cost calculation is fundamental to the MPEG algorithm, it is not surprising to find a custom operation—ume8uu—that implements this operation exactly. The definition of ume8uu operation is shown in Figure 8.

```
unsigned char A[16][16];
unsigned char B[16][16];
.
.
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(A[row][col+0] - B[row][col+0]);
        cost1 = abs(A[row][col+1] - B[row][col+1]);
        cost2 = abs(A[row][col+2] - B[row][col+2]);
        cost3 = abs(A[row][col+3] - B[row][col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
    }
}
```

Figure 6. Unrolled and Parallel version of Figure 5.

If we hope to use a custom operation that processes four pixel values simultaneously, we first need to create four parallel pixel computations. Also, to use the ume8uu operation, however, the code must access the arrays with 32-bit word pointers instead of with 8-bit byte pointers.

Figure 6 shows a parallel version of the code from Figure 5. By unrolling the loop and simply giving each computation its own cost variable and then summing the costs all at once, each cost computation is completely independent.

Figure 7 shows the loop recoded to access A[] and B[] as one-dimensional instead of as two-dimensional arrays. We take advantage of our knowledge of C-language array storage conventions in order to perform this code transformation. Recoding to use one-dimensional arrays prepares the code for the transformation to 32-bit array accesses.

Figure 7 also shows the loop of Figure 6 recoded to use ume8uu. Once again taking advantage of our knowledge of the C-language array storage conventions, the one-dimensional byte array is now accessed as a one-dimensional 32-bit-word array.

Of course, since we are now using one-dimensional arrays to access the pixel data, it is natural to use a single 'for' loop instead of two. Figure 9 shows this streamlined version of the code without the inner loop. Since C-language arrays are stored as a linear vector of values, we can simply increase the number of iterations of the outer loop from 16 to 64 to traverse the entire array.

The recoding and use of the ume8uu operation has resulted in a substantial improvement in the performance of the match-cost loop. In the original version, the code executed 1280 operations (including loads, adds, subtracts, and absolute values); in the restructured version, there are only 256 operations—128 loads, 64 ume8uu operations, and 64 additions. This is a factor of five reduction in the number of operations executed. Also, the overhead of the inner loop has been eliminated, further increasing the performance advantage.

```
unsigned char A[16][16];
unsigned char B[16][16];
.
.
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;
for (row = 0, rowoffset = 0; row < 16;
     row += 1, rowoffset += 4)
{
    for (col4 = 0; col4 < 4; col4 += 1)
        cost += UME8UU(IA[rowoffset + col4],
                       IB[rowoffset + col4]);
}
```

Figure 7. Using the custom operation ume8uu to speedup the loop of Figure 6 resulted in a performance speedup of about

```

ume8uu      Sum of absolute values of
            unsigned 8-bit differences

'C' function prototype:
unsigned int
ume8uu(unsigned int a, unsigned int b);

Function of ume8uu:
abs(zero_extto32(a<31:24>) - zero_extto32(b<31:24>)) +
abs(zero_extto32(a<23:16>) - zero_extto32(b<23:16>)) +
abs(zero_extto32(a<15:8>) - zero_extto32(b<15:8>)) +
abs(zero_extto32(a<7:0>) - zero_extto32(b<7:0>))

```

Figure 8. Custom Operation ume8uu

6.0 APPLICATIONS

TM-1 has the potential to be used in many multimedia applications and only few of them are discussed.

6.1 Video Teleconferencing/Digital White Board

Businesses are increasingly turning towards interactive computing as a means of becoming more efficient. Collaborative computing, for instance, involves sharing applications amongst multiple personal computers and multipoint video teleconferencing.

TM-1 is a single chip video teleconferencing solution that runs all current video codecs across all common transport mechanisms. This may also include H.324 (POTS), H.320 (ISDN) and H.323 (LAN).

6.2 Multimedia Card for Consumer Multimedia Applications

The achievement of true computer based realism is only possible with a fully integrated approach to multimedia -- one that permits the smooth flow of audio, video, graphics and communications. Today's computer user wants a highly interactive and realistic experience. The Trimedia processor makes this possible.

TM-1 is a low-cost, programmable processor for the consumer multimedia market. This product provides the additional processing power required for a true-to-life computer based experience. The Trimedia processor concurrently processes multiple data types including audio, video, graphics and communications. The first version of this chip, designated TM-1, is targeted for the PC market.

7.0 SUMMARY

The TM-1 is the first programmable multimedia processor from the Trimedia division of the Philips Semiconductors. The TM-1 has high performance VLIW CPU core, efficient 'C' compiler with multimedia library functions, glueless logic to high-bandwidth SDRAM, standard PCI bus interface, and standard interfaces to video and audio stream that make the TM-1 the next generation multimedia processor for stand-alone systems such as the video phone, video conferencing system and plug-in multimedia cards for the PC systems.

```

unsigned char A[16][16];
unsigned char B[16][16];

unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 1)
    cost += UME8UU(IA[i], IB[i]);

```

Figure 9. The loop of Figure 7 with the Inner loop eliminated.

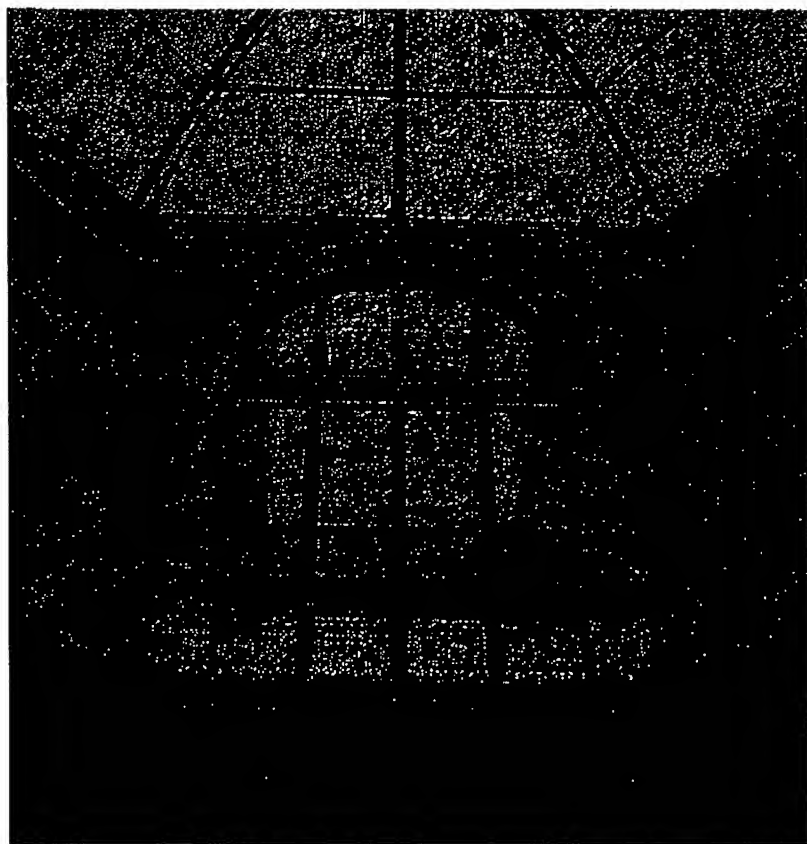
8.0 REFERENCES

- [1] J. Labrousse, G. A. Slavenburg. "A 50MHz Microprocessor with a VLIW Architecture." ISSCC, 1990.
- [2] J. Labrousse, G. A. Slavenburg. "CREATE-LIFE: A Design System for High Performances VLSI Circuits" ICCD-88. 1988.
- [3] J. Labrousse, G. A. Slavenburg. "CREATE-LIFE: A Modular Design Approach for High Performances ASIC's." Comcon Conference 1990.
- [4] Brian Case. "Philips Hopes to Displace DSPs with VLIW" Microprocessor Report, December 5, 1994.
- [5] Brian Case. "First Trimedia Chip Boards PCI Bus." Microprocessor Report, November 1995.
- [6] Gert Slavenburg. "The Trimedia VLIW-Based PCI Multimedia Processor" In Microprocessor Forum, October, 1995.
- [7] A.S. Huang, G. Slavenburg, J.P. Shen. "Speculative Disambiguation: A compilation Technique for Dynamic Memory Disambiguation". In 21st Annual International Symposium on Computer Architecture, April, 1994.
- [8] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, P.K. Rodman. "A VLIW Architecture for a Trace Scheduling Compiler." Proc. of ASPLOS II. October, 1987.
- [9] J.A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction." IEEE Trans on Computers. July 1981.
- [10] P.Y.T. Hsu and E.S. Davidson. "Highly Concurrent Scalar Processing." Proc. of the 13th Symposium on Computer Architecture, 1986

"The 'must-have' PC architecture reference set."

—PC Magazine's "Read Only" column

PENTIUM[®] PRO PROCESSOR SYSTEM ARCHITECTURE

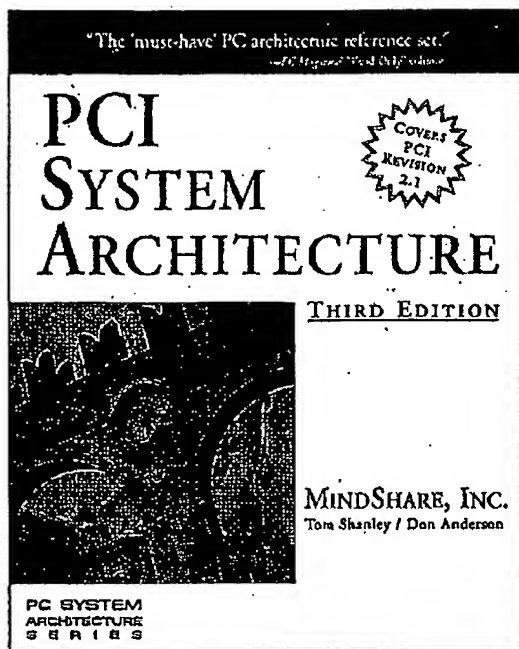


MINDSHARE, INC.

Tom Shanley

PC SYSTEM
ARCHITECTURE
S E R I E S

177AMD0061549



The PC System Architecture Series

by MindShare, Inc.

PCI System Architecture
Third Edition

ISA System Architecture
Third Edition

USB System Architecture

Pentium Pro Processor System Architecture

CardBus System Architecture

PCMCIA System Architecture
16-Bit PC Cards
Second Edition

Pentium™ Processor System Architecture
Second Edition

PowerPC™ System Architecture

Plug and Play System Architecture

80486 System Architecture
Third Edition

Protected Mode Software Architecture

EISA System
Second Edition

These titles are also available for bulk purchases by corporations, institutions, and other organizations in the U.S. For more information, please contact the Corporate, Government, and Special Sales Department at (800) 238-9682 or at cgss@aw.com.

Pentium® Pro Processor System Architecture

MINDSHARE, INC.

Tom Shanley



ADDISON-WESLEY DEVELOPERS PRESS

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California

Berkeley, California, • Don Mills, Ontario • Sydney

Bonn • Amsterdam • Tokyo • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designators appear in this book, and Addison-Wesley was aware of the trademark claim, the designations have been printed in initial capital letters or all capital letters.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-47953-2

Copyright ©1997 by MindShare, Inc.

A-W Developers Press is a division of Addison Wesley Longman, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor: Kathleen Tibbetts
Project Manager: Sarah Weaver
Cover Design: Barbara T. Atkinson
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9-MA-0099989796
First Printing, December 1996

Addison-Wesley books available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Find A-W Developers Press on the World-Wide Web at:
<http://www/aw.com/devpress/>

To my daughter Jennifer—I love you and I’m very proud of you.

Contents

About This Book

The MindShare Architecture Series	1
Cautionary Note	2
What This Book Covers	2
What This Book Does not Cover.....	2
Organization of This Book.....	2
Who This Book Is For.....	4
Prerequisite Knowledge	4
Documentation Conventions.....	4
Hexadecimal Notation	4
Binary Notation	4
Decimal Notation	5
Signal Name Representation	5
Warning.....	5
Identification of Bit Fields (logical groups of bits or signals)	6
Register Field References	6
Visit Our Web Page	6
We Want Your Feedback.....	6

Part 1: System Overview

Chapter 1: System Overview

Introduction.....	11
What Is a Cluster?	13
What Is a Quad or 4-Way System?	13
Bootstrap Processor.....	13
Starting Up Other Processors.....	13
Relationship of Processors to Main Memory	14
Processors' Relationships to Each Other	14
Host/PCI Bridges	17
Bridges' Relationship to Processors	17
Bridges' Relationship to PCI Masters and Main Memory	19
Bridges' Relationship to PCI Targets	19
Bridges' Relationship to EISA or ISA Targets.....	19
Bridges' Relationship to Each Other	20
Bridge's Relationship to EISA and ISA Masters and DMA	20

Contents

Part 2: Processor's Hardware Characteristics

Hardware Section 1: The Processor

Chapter 2: Processor Overview

Two Bus Interfaces.....	25
External Bus.....	26
Bus on Earlier Processors Inefficient for Multiprocessing.....	26
Pentium Bus has Limited Transaction Pipelining Capability	27
Pentium Pro Bus Tuned for Multiprocessing	28
IA = Legacy	29
Instruction Set.....	29
IA Instructions Vary in Length and are Complex.....	29
Pentium Pro Translates IA Instructions into RISC Instructions	29
In-Order Front End	30
Out-of-Order Middle.....	30
In-Order Rear End	30
Register Set.....	31
IA Register Set is Small	31
Pentium Pro has 40 General-Purpose Registers	32
Elimination of False Register Dependencies.....	33
Introduction to the Internal Architecture.....	34

Chapter 3: Processor Power-On Configuration

Features that are Automatically Configured.....	37
Setup and Hold Time Requirements.....	39
Run BIST Option	39
Error Observation Options	39
In-Order Queue Depth Selection	40
Power-On Restart Address Selection	40
FRC Mode Enable/Disable.....	40
APIC ID Selection	40
Selecting Tri-State Mode	41
Processor Core Speed Selection	41
Processor's Agent and APIC ID Assignment.....	42
FRC Mode	44
Program-Accessible Startup Features.....	45

Chapter 4: Processor Startup

Selection of Processor's Agent and APIC IDs	51
Processor's State After Reset	52
EDX Contains Processor Identification Info	55
State of Caches and Processor's Ability to Cache	55
Selection of Bootstrap Processor (BSP)	55
Introduction	55
BSP Selection Process	57
APIC Arbitration Background	57
Startup APIC Arbitration ID Assignment	57
BSP Selection Process	57
Processor's Initial Memory Reads	58
How APs are Started	59
Uni-Processor OS	59
SMP OS	59
AP Detection	60
AP Task Assignment	60

Chapter 5: The Fetch, Decode, Execute Engine

Please Note	61
Introduction	62
Enabling the Caches	64
Prefetcher	64
Issues Sequential Read Requests to Code Cache	64
Detailed Description of Prefetcher Operation	65
Brief Description of Pentium Pro Processor	67
Beginning, Middle and End	68
In-Order Front End	68
Out-of-Order (OOO) Middle	68
In-Order Rear End	68
Intro to the Instruction Pipeline	68
In-Order Front End	71
Instruction Fetch Stages	71
IFU1 Stage: 32-Byte Line Fetched from Code Cache	71
IFU2 Stage: Marking Boundaries and Dynamic Branch Prediction	71
IFU3 Stage: Align Instructions for Delivery to Decoders	72
Decode Stages	74
DEC1 Stage: Translate IA Instructions into Micro-Ops	74
Micro Instruction Sequencer (MIS)	74
DEC2 Stage: Move Micro-Ops to ID Queue	75

Contents

Queue Micro-Ops for Placement in Pool	75
Second Chance for Branch Prediction	76
RAT Stage: Overcoming the Small IA Register Set.....	76
ReOrder Buffer (ROB) Stage.....	77
Instruction Pool (ROB) is a Circular Buffer	77
Out-of-Order (OOO) Middle	80
In-Order Rear End (RET1 and RET2 Stages)	80
Three Scenarios	81
Scenario One: Reset Just Removed.....	81
Starvation!	81
First Instruction Fetch	82
First Memory Read Bus Transaction.....	82
Eight Bytes Placed in Prefetch Streaming Buffer	82
Instruction Boundaries Marked and BTB Checked	84
Between One and Three Instructions Decoded into Micro-Ops.....	84
Source Operand Location Selected (RAT).....	86
Micro-Ops Advanced to ROB and RS.....	87
Micro-Ops Dispatched for Execution	88
Micro-Ops Executed	88
Result to ROB Entry (and other micro-ops if necessary)	88
Micro-op Ready for Retirement?	88
Micro-Op Retired	89
Scenario Two: Processor's Caches Just Enabled.....	90
Scenario Three: Caches Enabled for Some Time	91
Memory Data Accesses—Loads and Stores	104
Handling Loads.....	104
Handling Stores.....	106
Description of Branch Prediction.....	108
486 Branch Handling	108
Pentium Branch Prediction.....	108
Pentium Pro Branch Prediction	108
Mispredicted Branches are VERY Costly!.....	108
Dynamic Branch Prediction	112
General	112
Yeh's Prediction Algorithm	112
Return Stack Buffer (RSB).....	112
Static Branch Prediction	113
Code Optimization	115
General.....	115
Reduce Number of Branches	115
Follow Static Branch Prediction Algorithm	116
Identify and Improve Unpredictable Branches	116

Contents

Don't Intermingle Code and Data	116
Align Data	116
Avoid Serializing Instructions	116
Where Possible, Do Context Switches in Software	117
Eliminate Partial Stalls: Small Write Followed by Full-Register Read.....	117
Data Segment Register Changes Serialize Execution	117

Chapter 6: Rules of Conduct	
The Problem.....	119
General.....	119
A Memory-Mapped IO Example.....	120
Pentium Solution	120
Pentium Pro Solution	121
State of the MTRRs after Reset	123
Memory Types	124
Uncacheable (UC) Memory	124
Write-Combining (WC) Memory	124
Write-Through (WT) Memory	126
Write-Protect (WP) Memory	126
Write-Back (WB) Memory	127
Rules as Defined by MTRRs	127
Rules of Conduct Provided in Bus Transaction.....	129
MTRRs and Paging: When Worlds Collide	129
Detailed Description of the MTRRs.....	131

Chapter 7: The Processor Caches	
Cache Overview.....	133
Introduction to Data Cache Features	134
Introduction to Code Cache Features	135
Introduction to L2 Cache Features	135
Introduction to Snooping.....	136
Determining Processor's Cache Sizes and Structures.....	137
L1 Code Cache.....	137
Code Cache Uses MESI Subset: S and I	138
Code Cache Contains Only Raw Code	138
Code Cache View of Memory Space.....	140
Code TLB (ITLB)	140
Code Cache Lookup	140
Code Cache Hit	141
Code Cache Miss	141
Code Cache LRU Algorithm: Make Room for the New Guy	141

Contents

Code Cache Castout	142
Code Cache Snoop Ports.....	142
L1 Data Cache	143
Data Cache Uses MESI Cache Protocol	144
Data Cache View of Memory Space.....	146
Data TLB (DTLB).....	146
Data Cache Lookup	146
Data Cache Hit	147
Relationship of L2 and L1 Caches	147
Relationship of L2 to L1 Code Cache.....	148
Relationship of L2 and L1 Data Cache.....	149
Read Miss on L1 and L2.....	149
Read Miss On All Other Caches	150
Read Hit On E or S Line in One or More Other Caches.....	150
Read Hit On Modified Line in One Other Cache	150
Write Hit On L1 Data Cache	150
Write Hit on S Line in Data Cache	150
Write Hit On E Line in Data Cache.....	151
Write Hit On M Line in Data Cache	151
Write Miss On L1 Data Cache.....	151
L1 Data Cache Castout.....	152
Data Cache LRU Algorithm: Make Room for the New Guy	152
Data Cache Pipeline.....	155
Data Cache is Non-Blocking.....	156
Earlier Processor Caches Blocked, but Who Cares?	156
Pentium Pro Data Cache is Non-Blocking, and That's Important!	157
Data Cache has Two Service Ports	157
Two Address and Two Data Buses	157
Simultaneous Load/Store Constraint.....	159
Data Cache Snoop Ports.....	161
Unified L2 Cache.....	162
L2 Cache Uses MESI Protocol	162
L2 Cache View of Memory Space.....	164
Request Received	164
L2 Cache Lookup	164
L2 Cache Hit	165
L2 Cache Miss.....	166
L2 Cache LRU Algorithm: Make Room for the New Guy	166
L2 Cache Pipeline.....	167
L2 Cache Snoop Ports.....	170
Toggle Mode Transfer Order.....	171
Self-Modifying Code and Self-Snooping	173

Contents

Description.....	173
Don't Let Your Data and Code Get Too Friendly!	176
ECC Error Handling.....	176

Hardware Section 2: Bus Intro and Arbitration

Chapter 8: Bus Electrical Characteristics

Introduction.....	179
Everything's Relative	180
All Signals Active Low.....	181
Powerful Pullups Snap Lines High Fast.....	182
The Layout.....	182
Synchronous Bus.....	183
Setup and Hold Specs	183
Setup Time	183
Hold Time	183
How High is High and How Low is Low?	184
After You See Something, You have One Clock to Do Something About It	185

Chapter 9: Bus Basics

Agents.....	187
Agent Types.....	187
Multiple Personalities.....	188
Uniprocessor vs. Multiprocessor Bus.....	189
Request Agents.....	191
Request Agent Types.....	191
Agent ID	191
What Agent ID Used For	191
How Agent ID Assigned.....	192
Transaction Phases.....	192
Pentium Transaction Phases.....	192
Pentium Pro Transaction Phases	192
Transaction Pipelining.....	193
Bus Divided into Signal Groups	193
Step One: Gain Ownership of Request Signal Group.....	193
Step Two: Issue Transaction Request.....	194
Step Three: Yield Request Signal Group, Proceed to Next Signal Group.....	194
Phases Proceed in Predefined Order.....	194
Request Phase.....	195
Error Phase.....	195
Snoop Phase.....	195

Contents

Response Phase	195
Data Phase.....	196
Next Agent Can't Use Signal Group Until Current Agent Done With It	196
Transaction Tracking.....	199
Request Agent Transaction Tracking.....	199
Snoop Agent Transaction Tracking	199
Response Agent Transaction Tracking	200
The IOQ	200

Chapter 10: Obtaining Bus Ownership

Request Phase	201
Symmetric Agent Arbitration—Democracy at Work	202
No External Arbiter Required	202
Agent ID Assignment	204
Arbitration Algorithm.....	204
Rotating ID.....	204
Busy/Idle State.....	204
Bus Parking.....	204
Be Fair!.....	205
What Signal Group are You Arbitrating For?.....	205
Requesting Ownership.....	205
Example of One Symmetric Agent Requesting Ownership	206
Example of Two Symmetric Agents Requesting Ownership	207
Definition of an Arbitration Event	209
Once BREQn# Asserted, Keep Asserted Until Ownership Attained	210
Example Case Where Transaction Cancelled Before Started	210
Bus Parking Revisited	210
Priority Agent Arbitration—Despotism	211
Example Priority Agents.....	211
Priority Agent Beats Symmetric Agents, Unless.....	213
Using Simple Approach, Priority Agent Suffers Penalty.....	214
Smarter Priority Agent Gets Ownership Faster	217
Ownership Attained in 2 BCLKs	217
Ownership Attained in 3 BCLKs	219
Be Fair to the Common People.....	221
Priority Agent Parking	221
Locking—Shared Resource Acquisition.....	221
Shared Resource Concept	221
Testing Availability and Gaining Ownership of Shared Resources	222
Race Condition Can Present Problem.....	222
Guaranteeing Atomicity of Read/Modify/Write	223
LOCK Instruction Prefix	225

Contents

Processor Automatically Asserts LOCK# for Some Operations	225
Use Locked RMW to Obtain and Give Up Semaphore Ownership	225
Duration of Locked Transaction Series.....	226
Locking a Cache Line	227
Advantage of Cache Line Locking	227
New Directory Bit—Cache Line Locked	228
Read and Invalidate Transaction (RWITM, or Kill).....	228
Line in E or M State	228
Semaphore Not in Processor's L1 or L2 Cache.....	229
Semaphore in Cache in E State	230
Semaphore in Cache in S State.....	230
Semaphore in Cache in M State	230
Blocking New Requests—Stop! I'm Full!.....	230
BNR# is Shared Signal.....	231
Stalled/Throttled/Free Indicator	231
Open Gate, Let One Out, Close Gate	232
Open Gate, Leave It Open, Let Them All Out	232
Gate Wide Open and then Slammed Shut	233
BNR# Behavior at Powerup.....	233
BNR# and the Built-In Self-Test (BIST).....	234
BNR# Behavior During Runtime.....	236

Hardware Section 3: The Transaction Phases

Chapter 11: The Request and Error Phases

Caution	241
Request Phase	242
Introduction to the Request Phase.....	242
Request Signal Group is Multiplexed	243
Introduction to the Transaction Types.....	243
Contents of Request Packet A	245
32-bit vs. 36-bit Addresses	248
Contents of Request Packet B.....	249
Error Phase.....	253
In-Flight Corruption	253
Who Samples AERR#?	255
Request Agent	255
Other Bus Agents.....	255
Who Drives AERR#?.....	255
Request Agent's Response to AERR# Assertion	255
Other Guys are Very Polite.....	256

Contents

Chapter 12: The Snoop Phase

Agents Involved in Snoop Phase	257
Snoop Phase Has Two Purposes	260
Snoop Result Signals are Shared, DEFER# Isn't.....	260
Snoop Phase Duration Is Variable	260
Is There a Snoop Stall Duration Limit?	263
Memory Transaction Snooping.....	263
Snoop's Effects on Caches.....	263
After Snoop Stall, How Soon Can Next Snoop Result be Presented?	267
Self-Snooping.....	268
Non-Memory Transactions Have a Snoop Phase	268
Transaction Retry and Deferral.....	268
Permission to Defer Transaction Completion.....	268
DEFER# Assertion Delays Transaction Completion.....	269
Transaction Retry	269
Transaction Deferral	270
Mail Delivery Analogy.....	270
Example System Operation Overview	270
The Wrong Way.....	270
The Right Way	272
Bridge Should be a Faithful Messenger.....	273
Detailed Deferred Transaction Description.....	274
What if HITM# and DEFER# both Asserted?	274
How Does Locking Change Things?	276

Chapter 13: The Response and Data Phases

Note on Deferred Transactions	277
Purpose of Response Phase.....	277
Response Phase Signal Group.....	278
Response Phase Start Point	279
Response Phase End Point	279
List of Responses.....	279
Response Phase May Complete Transaction.....	281
Data Phase Signal Group.....	281
Five Example Scenarios.....	282
Transaction that Doesn't Transfer Data.....	282
Read that Doesn't Hit a Modified Line and is Not Deferred.....	284
Basics.....	284
Detailed Description.....	286
How Does Response Agent Know Transfer Length?	287

Contents

What's the Earliest that DBSY# Can be Deasserted?	287
Relaxed DBSY# Deassertion:.....	287
Write that Doesn't Hit a Modified Line and Isn't Deferred.....	287
Basics.....	289
Previous Transaction May Involve a Write	289
Earliest TRDY# Assertion is 1 Clock After Previous Response Issued.....	289
When Does Request Agent First Sample TRDY#?	289
When Does Request Agent Start Using Data Bus?	290
When Can TRDY# Be Deasserted?	290
When Does Request Agent Take Ownership of Data Bus?	291
Deliver the Data	291
On AERR# or Hard Failure Response.....	291
Snoop Agents Change State of Line from E->I or S->I.....	291
Read that Hits a Modified Line.....	291
Basics.....	292
Transaction Starts as a Read from Memory	294
From Memory Standpoint, Changes from Read to Write.....	294
Memory Asserts TRDY# to Accept Data	294
Memory Must Drive Response At Right Time.....	294
Snoop Agent Asserts DBSY# and Memory Drives Response	295
Snoop Agent Supplies Line to Memory and to Request Agent	295
Snoop Agent Changes State of Line from M->S.....	295
Write that Hits a Modified Line.....	295
Data Phase Wait States.....	298
Special Case—Single Quadword, 0-Wait State Transfer	301
Response Phase Parity.....	303

Hardware Section 4: Other Bus Topics

Chapter 14: Transaction Deferral

Introduction to Transaction Deferral	307
Example System Model.....	307
Typical PC Server Model.....	309
The Problem.....	309
Possible Solutions	310
An Example Read	311
Read Transaction Memorized and Deferred Response Issued	311
Bridge Performs PCI Read Transaction.....	313
Deferred Reply Transaction Issued.....	313
Original Request Agent Selected	317
Bridge Provides Snoop Result	317
Response Phase—Role Reversal.....	317

Contents

Data Phase.....	318
Trackers Retire Transaction.....	318
Other Possible Responses.....	318
An Example Write.....	320
Transaction and Write Data Memorized, Deferred Response Issued.....	321
PCI Transaction Performed and Data Delivered to Target.....	323
Deferred Reply Transaction Issued.....	323
Original Request Agent Selected.....	325
Bridge Provides Snoop Result.....	325
Response Phase—Role Reversal.....	325
There is No Data Phase.....	326
Trackers Retire Transaction.....	326
Other Possible Responses.....	326
Pentium Pro Support for Transaction Deferral.....	327

Chapter 15: IO Transactions

Introduction.....	329
IO Address Range.....	330
Data Transfer Length.....	330
Behavior Permitted by Specification.....	330
How Pentium Pro Processor Operates.....	331

Chapter 16: Central Agent Transactions

Point-to-Point vs. Broadcast.....	333
Interrupt Acknowledge Transaction.....	334
Background.....	334
How Pentium Pro is Different.....	337
Host/PCI Bridge is Response Agent.....	337
Special Transaction.....	338
General.....	338
Message Types.....	338
Branch Trace Message Transaction Used for Program Debug.....	340
What's the Problem?.....	340
What's the Solution?.....	341
Enabling Branch Trace Message Capability.....	341
Branch Trace Message Transaction.....	342
Packet A Composition.....	342
Packet B Composition.....	342
Proper Response.....	343
Data Composition.....	343

Chapter 17: Other Signals

Error Reporting Signals	345
Bus Initialize (BINIT#).....	345
Description.....	345
Assertion/Deassertion Protocol	346
Bus Error (BERR#).....	346
Description.....	346
BERR#/BINIT# Assertion/Deassertion Protocol.....	347
Internal Error (IERR#)	347
Functional Redundancy Check Error (FRCERR)	347
PC-Compatibility Signals.....	348
A20 Mask (A20M#)	348
FERR# and IGNNE#	348
Diagnostic Support Signals	349
Interrupt-Related Signals	350
Processor Present Signals	352
Power Supply Pins.....	352
Miscellaneous Signals.....	354

Part 3: Processor's Software Characteristics

Chapter 18: Instruction Set Enhancements

Introduction.....	359
CPUID Instruction Enhanced	359
Before Executing, Determine if Supported	359
Basic Description.....	361
Vendor ID and Max Input Value Request.....	361
Request for Vendor ID String and Max EAX Value.....	362
Request for Version and Supported Features.....	362
Request for Cache and TLB Information	364
CPUID is a Serializing Instruction.....	366
Serializing Instructions Impact Performance.....	367
Conditional Move (CMOV) Eliminates Branches	367
Conditional FP Move (FCMOV) Eliminates Branches	367
FCOMI, FCOMIP, FUCOMI, and FUCOMIP	368
Read Performance Monitoring Counter (RDPMC)	368
What's RDPMC Used For?	368
Who Can Execute RDPMC?	368
RDPMC Not Serializing Instruction.....	369
RDPMC Description.....	369

Contents

Read Time Stamp Counter (RDTSC)	370
What's RDTSC Used For	370
Who Can Execute RDTSC?	370
RDTSC Doesn't Serialize	370
RDTSC Description	371
My Favorite—UD2	371
Accessing MSRs	371
Testing for Processor MSR Support	371
Causes GP Exception If	371
Input Parameters	372

Chapter 19: Register Set Enhancements

New Registers	373
Introduction	373
DebugCTL, LastBranch and LastException MSRs	374
Introduction	374
Last Branch, Interrupt or Exception Recording	375
Single-Step Exception on Branch, Exception or Interrupt	376
New Bits in Pre-Existent Registers	376
CR4 Enhancements	376
CR3 Enhancements	377

Chapter 20: Paging Enhancements

Background on Paging	379
Page Size Extension (PSE) Feature	380
The Problem	380
The Solution—Big Pages	381
How It Works	381
Physical Address Extension (PAE) Feature	383
How Paging Normally Works	383
What Is the PAE?	385
How Is the PAE Enabled?	385
Changes to the Paging-Related Data Structures	385
Programmer Still Restricted to 32-bit Addresses and 2 ²⁰ Pages	387
Pages Can be Distributed Throughout Lower 64GB	387
CR3 Contains Base Address of PDPT	387
Format of PDPT Entry	388
Format of Page Directory Entry	390
Format of Page Table Entry	392
The PAE and the Page Size Extension (PSE)	394
Global Page Feature	398

Contents

The Problem	398
The Solution	398

Chapter 21: Interrupt Enhancements	
New Exceptions	402
Added APIC Functionality	402
VM86 Mode Extensions	403
- VM86 Mode Background	403
Interrupt-Related Problems and VM86 Tasks	403
Software Overhead Associated with CLI/STI Execution	404
Attempted Execution of CLI by VM86 Task	404
Attempted Execution of STI Instruction	405
Servicing of Software Interrupts by DOS or OS	406
Solution—VM86 Mode Extensions	406
Introduction	406
CLI/STI Solution	409
EFLAGS[VIF] = 1, EFLAGS[IF] = 1, Interrupt Occurs	409
EFLAGS[VIF] = 0, EFLAGS[IF] = 1, Interrupt Occurs	409
Software Interrupt Redirection Solution	410
Virtual Interrupt Handling in Protected Mode	413

Chapter 22: Machine Check Architecture	
Purpose of Machine Check Architecture	415
Machine Check Architecture in the Pentium Processor	416
Testing for Machine Check Support	417
Machine Check Exception	419
Machine Check Architecture Register Set	419
Composition of Global Register Set	421
MCG_CAP Register	421
MCG_STATUS Register	421
MCG_CTL Register	422
Composition of Each Register Bank	423
General	423
MCI_STATUS Register	424
MSR Addresses of the Machine Check Registers	426
Initialization of Register Set	428
Machine Check Architecture Error Format	429
Simple Error Codes	429
Compound Error Codes	430
External Bus Error Interpretation	433

Contents

Chapter 23: Performance Monitoring and Timestamp

Time Stamp Counter Facility	438
Time Stamp Counter (TSC) Definition	438
Detecting Presence of the TSC	438
Accessing the Time Stamp Counter	438
Reading the TSC Using RDTSC Instruction	438
Reading the TSC Using RDMSR Instruction	439
Writing to the TSC	439
Performance Monitoring Facility	439
Purpose of the Performance Monitoring Facility	439
Performance Monitoring Registers	439
PerfEvtSel0 and PerfEvtSel1 MSRs	440
PerfCtr0 and PerfCtr1	441
Accessing the Performance Monitoring Registers	442
Accessing the PerfEvtSel MSRs	442
Accessing the PerfCtr MSRs	442
Accessing Using RDPMC Instruction	442
Accessing Using RDMSR/WRMSR Instructions	442
Event Types	443
Starting and Stopping the Counters	443
Starting the Counters	443
Stopping the Counters	443
Performance Monitoring Interrupt on Overflow	443

Chapter 24: MMX: Matrix Math Extensions

Please Note	445
Problems Addressed by MMX	445
Problem: Math on Packed Bytes/Words/Dwords	445
Solution: MMX Matrix Math/Logical Operations	446
Problem: Data not Packed	447
Solution: MMX Pack and Unpack Instructions	447
Problem: Math Overflows/Underflows	447
Solution: Saturating Math	448
Problem: Comparisons and Branches	448
Solution: MMX Parallel Comparisons	448
Single Instruction, Multiple Data (SIMD)	451
Detecting Presence of MMX	451
Changes to Programming Environment	451
General	451
Handling a Task Switch	453

Contents

When Exiting MMX Routine, Execute EMMS	453
MMX Instruction Set	453
Instruction Groups	453
Instruction Syntax	454
Instruction Set	454
Pentium Pro MMX Execution Units	456

Part 4: Overview of Intel Pentium Pro Chipsets

Chapter 25: 450GX and KX Chipsets

Processor Bus Operation	461
PCI Bus Operation	461
450GX Chipset	461
Overview	461
Major Features	462
Overview of Compatibility PB	466
Compatibility PB is the Target	466
Transaction Initiated on Processor Bus	466
Transaction Initiated on PCI Bus	466
Target on Other Side of Compatibility PB	467
Transaction Initiated on Processor Bus	467
Transaction Initiated on PCI Bus	467
Neither Compatibility PB nor Device on Other Side Is Target	467
Overview of Aux PB	467
Aux PB is the Target	468
Transaction Initiated on Processor Bus	468
Transaction Initiated on PCI Bus	468
Target on Other Side of Aux PB	468
Transaction Initiated on Processor Bus	468
Transaction Initiated on PCI Bus	468
Neither Aux PB nor Device on Other Side Is Target	469
Overview of Memory Controller	469
Startup Autoconfiguration	469
Introduction	469
Autoconfiguration of PBs	469
Autoconfiguration of Memory Controller	472
Processor Bus Agent Configuration	473
Transaction Deferral Not Implemented—Retry Used Instead	475
How Chipset Members are Configured by Software	476
Chipset Configuration Mechanism	476
PB Configuration Registers	478

Contents

Memory Controller Configuration Registers.....	489
450KX Chipset.....	494
Overview	494
Major Features	494

Chapter 26: 440FX Chipset

Processor Bus Operation.....	497
PCI Bus Operation	497
ChipSet Overview	497
Major Features	498
PMC Configuration Registers	499

Appendix: The MTRR Registers

Introduction.....	505
Feature Determination	507
MTRRdefType Register.....	508
Fixed-Range MTRRs.....	509
Enabling the Fixed-Range MTRRs.....	509
Define Rules Within 1st MB	510
Variable-Range MTRRs.....	512
Enabling the Variable-Range MTRRs	512
Number of Variable-Range MTRRs	512
Format of Variable-Range MTRR Register Pairs.....	512
MTRRphysBasen Register	513
MTRRphysMaskn Register.....	513
Examples	513
Index	513

Figures

1-1	Block Diagram of a Typical Server	12
2-1	Two Bus Interfaces	26
2-2	Pentium Transaction Pipelining	28
2-3	IA General Register Set	32
2-4	Simplified Processor Block Diagram.....	35
3-1	Automatic Feature Determination	38
3-2	Processor Agent ID Assignment.....	44
3-3	EBL_CR_POWERON MSR.....	50
4-1	EDX Contents after Reset.....	55
4-2	System Block Diagram	56
5-1	Overall Processor Block Diagram	63
5-2	Example Instructions in Memory	66
5-3	Contents of Prefetch Streaming Buffer Immediately after Line Fetched.....	66
5-4	The Three Instruction Decoders.....	67
5-5	Instruction Pipeline.....	69
5-6	The Three Instruction Decoders.....	73
5-7	Decoders and the Micro Instruction Sequencer (MIS).....	75
5-8	The ROB, the RS and the Execution Units.....	79
5-9	Scenario One Example—Reset Just Removed	83
5-10	The IFU2 Stage.....	84
5-11	Decoders and the Micro Instruction Sequencer (MIS).....	86
5-12	The ReOrder Buffer (ROB) and The Reservation Station (RS)	87
5-13	Code Cache and L2 Cache Miss.....	90
5-14	Scenario Three	92
5-15	Micro-Ops in Entries 13, 14 and 15 Have Been Retired.....	102
5-16	Micro-Ops in Entries 16, 17 and 18 Have Been Retired.....	103
5-17	Load and Store Execution Units	105
5-18	Load Pipeline Stages.....	105
5-19	Fetch/Decode/Execute Engine	111
5-20	Static Branch Prediction Algorithm	114
5-21	Static Branch Prediction Logic	115
6-1	Fixed-Range MTRRs Define Rules of Conduct within First MB of Memory Space.....	122
6-2	MTRRdefType Register.....	123
7-1	Processor Cache Overview	134
7-2	The L1 Code Cache	139
7-3	L1 Data Cache.....	145
7-4	Relationship of L2 to L1 Caches.....	148
7-5	L1 Data Cache.....	154
7-6	Cache Pipeline Stages	156
7-7	Example of Pipelined Data Cache Accesses.....	156
7-8	Data and Address Bus Interconnect between Data Cache and Execution Units.....	158
7-9	Data Cache Structure.....	160
7-10	256KB L2 Cache.....	163
7-11	Data Path between L2 and L1 Caches.....	165
7-12	The L1 Data and L2 Cache Pipeline Stages	168
7-13	Example of Pipelined L2 Cache Accesses Resulting in Hits.....	169

Figures

7-14	2-Way Interleaved Memory Architecture	173
7-15	Processor Cache Overview	175
8-1	Each GTL Input Uses a Comparator	180
8-2	Logic Levels	181
8-3	GTL Bus Layout Basics.....	182
8-4	Setup and Hold Specs.....	184
8-5	Example.....	185
9-1	Block Diagram of a Typical Server System	190
9-2	Bus Signal Groups.....	193
9-3	Pipelined Transactions	198
10-1	System Block Diagram	203
10-2	Symmetric Agent Arbitration Signals.....	206
10-3	Example of One Processor Requesting Ownership	207
10-4	Example of Two Symmetric Agents Requesting Ownership	209
10-5	System Block Diagram	212
10-6	Simple Approach Results in Penalty.....	216
10-7	Example Where Priority Agent Attains Ownership in 2 Clocks	218
10-8	Example Where Priority Agent Attains Ownership in 3 Clocks	220
10-9	Example of Symmetric Agent Performing Locked Transaction Series	224
10-10	Stalled/Throttled/Free Indicator States.....	232
10-11	BNR# at Powerup or After Reset.....	235
10-12	BNR# During Runtime	237
11-1	Two Information Packets Broadcast during Request Phase.....	242
11-2	Error Phase.....	254
12-1	System Block Diagram	259
12-2	Snoop Phase.....	261
12-3	System with Four Processors and Two Host/PCI Bridges.....	271
12-4	Two Processor Clusters Linked by Cluster Bridge	275
13-1	Transaction That Doesn't Require Data Transfer.....	284
13-2	Read that Doesn't Hit a Modified Line and Isn't Deferred	285
13-3	Write that Doesn't Hit a Modified Line and Isn't Deferred	288
13-4	Read that Hits a Modified Line.....	293
13-5	Write that Hits a Modified Line.....	297
13-6	Example of Two-Quadword Read with Wait States.....	299
13-7	Example of Four Quadword Write with Wait States.....	300
13-8	Example of Single-Quadword, 0-Wait State Write	302
14-1	Example System with one Pentium Pro Bus and Two Host/PCI Bridges	308
14-2	Read Transaction Receives a Deferred Response.....	312
14-3	Deferred Reply Transaction for Read	315
14-4	Write Transaction Receives Deferred Response.....	322
14-5	Deferred Reply Transaction for Write	324
16-1	Typical System Block Diagram	336
16-2	DEBUGCTL MSR.....	342
18-1	EFLAGS Register.....	360
18-2	Processor Version and Features Supported Information.....	363
18-3	CR4.....	369

Figures

19-1	DebugCTL MSR	375
19-2	CR4 Feature Bits	377
20-1	CR4	382
20-2	Page Directory 32-bit Entry Format (with PAE disabled)	382
20-3	Page Directory Entry for 4MB Page	383
20-4	Paging Directory Structure when PAE Enabled	386
20-5	CR3 Contains Base Address of PDPT	388
20-6	Format of a PDPT Entry	389
20-7	Format of Page Directory Entry that Points to Page Table	391
20-8	Format of Page Table Entry	393
20-9	Translation of Linear Address to a 2MB Page	396
20-10	Format of a Page Directory Entry for 2MB Page	397
21-1	Format of the Performance Counter LVT Entry	403
21-2	CR4	407
21-3	EFLAGS Register	408
21-4	Task State Segment (TSS)	411
22-1	CR4	417
22-2	Feature Support Information Returned by CPUID	418
22-3	Pentium Pro Machine Check Architecture Registers	420
22-4	MCG_CAP (Global Count and Present) Register	421
22-5	MCG_STATUS Register	422
22-6	MCI_ADDR Register	424
22-7	MCI_STATUS Register for Error Logging Bank i	424
22-8	MCI_STATUS Register Bit Assignment	429
23-1	PerfEvtSel0 and PerfEvtSel1 MSR Bit Assignment	441
24-1	Example MMX Packed-Byte Add	447
24-2	Results of PCMPEQW (Packed Compare If Words Equal) Instruction	449
24-3	Results of PANDN (Packed Logical AND NOT) Instruction	450
24-4	Result of PAND (Packed Logical AND) Instruction	450
24-5	Result of POR (Packed Logical OR) Instruction	451
24-6	MMX Registers are Mapped Over FP Registers	452
24-7	Pentium Pro MMX Execution Unit Distribution	457
25-1	Block Diagram of Typical System Designed Using Intel 450GX Chipset	464
25-2	Two Pentium Pro Clusters Connected by a Cluster Bridge	465
25-3	Pins Sampled on Trailing-edge of RESET#	474
25-4	TRC Register Bit Assignment	475
25-5	PCI Configuration Address Port	478
25-6	PB Configuration Registers	488
25-7	MC Configuration Registers	493
25-8	Typical Platform Designed Around 450KX Chipset	495
26-1	Typical Platform Designed Around 440FX Chipset	504
A-1	MTRRcap Register	508
A-2	MTRRdefType Register	509
A-3	First MB of Memory Space	511
A-4	Format of Variable-Range MTRRphysBasen Register	512
A-5	Format of MTRRphysMaskn Register	512

Tables

1-1	Example Cache State Transitions	15
1-2	Processor-Initiated Transactions	17
1-3	PCI Master-Initiated Transactions.....	19
3-1	Core Speed Selection	42
3-2	Processor Agent ID Assignment.....	43
3-3	Bit Assignment of EBL_CR_POWERON MSR.....	46
4-1	Effects of Reset on CPU.....	52
5-1	Pipeline Stages.....	69
5-2	Examples of IA Instruction Delivery to the Three Decoders.....	73
5-3	Basic Description of a ROB Entry	93
5-4	Description of Figure 5-14 on page 92 ROB Entries.....	94
5-5	Load Pipeline Stages	106
6-1	Memory Type Determination Using MTRRs	128
6-2	Basic Relationship of the MTRRs to Paging.....	129
6-3	Type if Paging and MTRRs enabled and Address Covered by Both.....	130
7-1	Results of External Snoop Presented to Code Cache.....	143
7-2	Results of External Snoop Presented to Data Cache.....	161
7-3	Results of External Snoop Presented to L2 Cache.....	170
7-4	Toggle Mode Quadword Transfer Order.....	172
10-1	Possible Cases Involving Priority Agent Arbitration.....	215
11-1	Packet B Signal Names.....	245
11-2	Currently-Defined Transaction Types.....	246
11-3	Request Packet A	248
11-4	Request Types (note: 0 = signal inactive, 1 = signal active).....	249
11-5	Data Transfer Length Field (see Table 11-4 on page 249)	250
11-6	Address Space Size Field (see Table 11-4 on page 249).....	250
11-7	Request Packet B	252
11-8	Attribute Field—Rules of Conduct (see Table 11-7 on page 252)	253
11-9	Deferred ID Composition (see Table 11-7 on page 252).....	253
11-10	Messages Broadcast Using Special Transaction (see Table 11-7 on page 252)	254
11-11	Extended Function Field (see Table 11-7 on page 252)	254
12-1	Snoop Result Table (0 = deasserted, 1 = asserted)	264
12-2	Effects of Snoop.....	266
13-1	Response List (0 = deasserted, 1 asserted)	281
13-2	Data Phase-related Signals	283
14-1	Request Packets A and B Content in Deferred Reply Transaction.....	318
14-2	PCI Read Transaction Completion and Deferred Reply Completion.....	321
14-3	PCI Write Transaction Completion and Deferred Reply Completion.....	328
16-1	Message Types	341
17-1	Diagnostic-Support Signals	351
17-2	Probe Port Signals.....	352
17-3	Interrupt-Related Signals.....	352

Tables

17-4	Processor Presence Signals	354
17-5	Processor Power-related Pins.....	355
17-6	Voltage ID Encoding	355
17-7	Miscellaneous Signals	356
18-1	Processor Type Field	365
18-2	Currently-Defined Cache/TLB Descriptors	366
18-3	Descriptors Returned by One of the Current Processor Implementations	367
21-1	VM86 Interrupt/Exception Handling	414
22-1	MCI_STATUS Register Bit Assignment.....	427
22-2	MSR Addresses of the Machine Check Registers.....	429
22-3	Simple Error Codes.....	432
22-4	Forms of Compound Error Codes.....	433
22-5	Transaction Type Sub-Field (TT).....	433
22-6	Memory Hierarchy Sub-Field (LL).....	433
22-7	Request Sub-Field (RRRR)	434
22-8	Definition of the PP, T, and II Fields	434
22-9	MCI_STATUS Breakdown for Bus-related Errors	435
23-1	PerfEvtSel0 and PerfEvtSel1 MSR Bit Assignment.....	442
24-1	MMX Instruction Set	457
25-1	PB Powerup ID Assignment	472
25-2	Default Startup Responsibilities of Compatibility PB	472
25-3	Default Startup Responsibilities of Aux PB	473
25-4	Memory Controller Powerup ID Assignment.....	474
25-5	Default Startup Responsibilities of Memory Controller 0	474
25-6	Default Startup Responsibilities of Memory Controller 1	475
25-7	PB Configuration Registers	480
25-8	MC Configuration Registers	491
26-1	PMC Configuration Register.....	501
A-1	Fixed-Range MTRRs.....	508

Acknowledgments

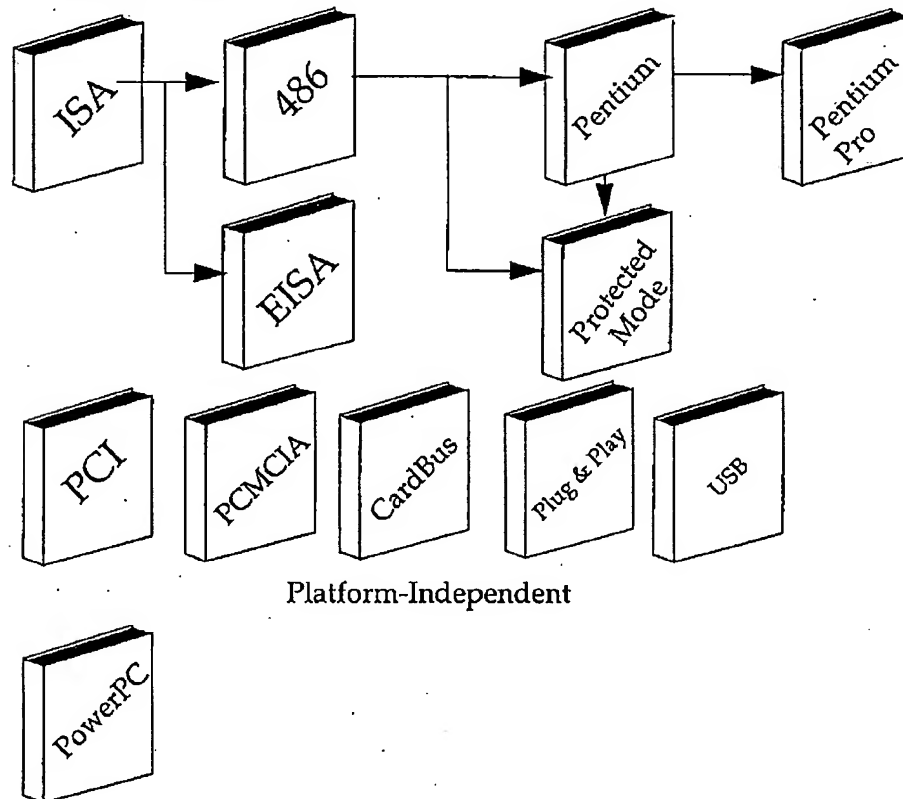
Kathleen Tibbetts, our editor at Addison-Wesley, remains calm and reasonable when faced with all manner of calamity: repeatedly slipped schedules, unreadable files, an author threatening suicide, etc. For her calming influence, I remain thankful.

About This Book

The MindShare Architecture Series

The MindShare Architecture book series includes: *ISA System Architecture*, *EISA System Architecture*, *80486 System Architecture*, *PCI System Architecture*, *Pentium System Architecture*, *PCMCIA System Architecture*, *PowerPC System Architecture*, *Plug-and-Play System Architecture*, *CardBus System Architecture*, *Protected Mode Software Architecture*, *USB System Architecture*, and *Pentium Pro Processor System Architecture*. The book series is published by Addison-Wesley.

Rather than duplicating common information in each book, the series uses the building-block approach. *ISA System Architecture* is the core book upon which most of the others build. The figure below illustrates the relationship of the books to each other.



Pentium Pro Processor System Architecture

Cautionary Note

The reader should keep in mind that MindShare's book series often deals with rapidly-evolving technologies. This being the case, it should be recognized that each book is a "snapshot" of the state of the targeted technology at the time that the book was completed. We attempt to update each book on a timely basis to reflect changes in the targeted technology, but, due to various factors (waiting for the next version of the spec to be "frozen," the time necessary to make the changes, and the time to produce the books and get them out to the distribution channels), there will always be a delay.

What This Book Covers

The purpose of this book is to provide a detailed description of the Pentium Pro processor both from the hardware and the software perspectives. As with our other x86 processor books, this book builds upon and does not duplicate information provided in our books on the previous generation processors. As an example, our *Pentium Processor System Architecture* book provided a detailed description of the APIC module, while this book only describes differences between the two implementations.

What This Book Does not Cover

This book does not describe the x86 instruction repertoire. There are a host of books on the market that already provide this information. It does, however, describe the new instructions added to the instruction set.

Organization of This Book

Pentium Pro Processor System Architecture extends MindShare's coverage of x86 processor architecture to the Pentium Pro. The author considers this book to be a companion to the MindShare books entitled *80486 System Architecture*, *Pentium Processor System Architecture*, and *Protected Mode Software Architecture* (published by Addison-Wesley). The book is organized as follows:

About This Book

- Part 1: System Overview
 - Chapter 1: System Overview
- Part 2: Processor's Hardware Characteristics
 - Hardware Section 1: The Processor
 - Chapter 2: Processor Overview
 - Chapter 3: Processor Power-On Configuration
 - Chapter 4: Processor Startup
 - Chapter 5: The Fetch, Decode, Execute Engine
 - Chapter 6: Rules of Conduct
 - Chapter 7: The Processor Caches
 - Hardware Section 2: Bus Intro and Arbitration
 - Chapter 8: Bus Electrical Characteristics
 - Chapter 9: Bus Basics
 - Chapter 10: Obtaining Bus Ownership
 - Hardware Section 3: The Transaction Phases
 - Chapter 11: The Request and Error Phases
 - Chapter 12: The Snoop Phase
 - Chapter 13: The Response and Data Phases
 - Hardware Section 4: Other Bus Topics
 - Chapter 14: Transaction Deferral
 - Chapter 15: IO Transactions
 - Chapter 16: Central Agent Transactions
 - Chapter 17: Other Signals
- Part 3: Processor's Software Characteristics
 - Chapter 18: Instruction Set Enhancements
 - Chapter 19: Register Set Enhancements
 - Chapter 20: Paging Enhancements
 - Chapter 21: Interrupt Enhancements
 - Chapter 22: Machine Check Architecture
 - Chapter 23: Performance Monitoring and Timestamp
 - Chapter 24: MMX: Matrix Math Extensions
- Part 4: Overview of Intel Pentium Pro Chipsets
 - Chapter 25: 450GX and KX Chipsets
 - Chapter 26: 440FX Chipset

Pentium Pro Processor System Architecture

Who This Book Is For

This book is intended for use by hardware and software design and support personnel. Due to the clear, concise explanatory methods used to describe each subject, personnel outside of the design field may also find the text useful.

Prerequisite Knowledge

It is highly recommended that the reader have a good knowledge of x86 processor architecture. Detailed descriptions of the 286 and 386 processors can be found in the MindShare book entitled *ISA System Architecture*. Detailed descriptions of the 486 and Pentium processors can be found in the MindShare books entitled *80486 System Architecture* and *Pentium Processor System Architecture*, respectively. Detailed descriptions of both real and protected mode operation can be found in the MindShare book entitled *Protected Mode Software Architecture*. All of these books are published by Addison-Wesley.

Documentation Conventions

This document utilizes the following documentation conventions for numeric values.

Hexadecimal Notation

All hex numbers are followed by an "h." Examples:

9A4Eh
C100h

Binary Notation

All binary numbers are followed by a "b." Examples:

0001 0101b
01b

Decimal Notation

Numbers without any suffix are decimal. When required for clarity, decimal numbers are followed by a "d." The following examples each represent a decimal number:

16
255
256d
128d

Signal Name Representation

Each signal that assumes the logic low state when asserted is followed by a pound sign (#). As an example, the HITM# signal is asserted low when a snoop agent has a hit on a modified line in its caches.

Signals that are not followed by a pound sign are asserted when they assume the logic high state.

Warning

The majority of the processor's signal pins are active low signals (e.g., all of the pins involved in a transaction). All tables in the *Intel Pentium Pro Volume One* data book, however, represent an asserted signal (in other words, in the electrically low state) with a one, while deasserted signals (electrically high) are represented by a zero in table entries. In other words, a "logical" one in a table indicates that the respective signal pin is asserted (in the electrically low state).

As an example, when a table entry indicates a one for the HITM# signal state, this indicates that it is asserted (electrically low).

Pentium Pro Processor System Architecture

Identification of Bit Fields (logical groups of bits or signals)

All bit fields are designated in little-endian bit ordering as follows:

[X:Y],

where "X" is the most-significant bit and "Y" is the least-significant bit of the field. As an example, the IOPL field in the EFLAGS register consists of bits [13:12], where bit 13 is the most-significant and bit 12 the least-significant bit of the field.

Register Field References

Bit fields in registers are frequently referred to using the form Reg[field name]. As an example, the reference CR4[DE] refers to the Debug Extensions bit in Control Register 4.

Visit Our Web Site

Our Web site contains a listing of all of our courses and books. In addition, it contains errata for a number of the books, a hot link to our publisher's web site, as well as course outlines.

www.mindshare.com

Our publisher's web page contains a listing of our currently-available books and includes pricing and ordering information. Their home page is accessible at:

www.aw.com/devpress

We Want Your Feedback

MindShare values your comments and suggestions. You can contact us via mail, phone, fax or internet email.

About This Book

Phone: (972) 231-2216, and, in the U.S., (800) 633-1440

Fax: (972) 783-4715

E-mail: tshanley@interserv.com

For information on MindShare seminars, check our web site.

Mailing Address:

MindShare, Inc.
2202 Buttercup Drive
Richardson, Texas 75082

Part 1:

System Overview

This Part

Part 1 of the book presents a basic description of a typical server's architecture, describing the relationships of the processors, caches, main memory, host/PCI bridges, PCI masters and targets, and E/ISA masters and targets. This subject is covered first to provide a backdrop for the subjects covered in the remainder of the book.

The Next Part

Part 2 of the book focuses on the hardware aspects of the processor's internal operation as well as its bus structure and protocol.

1

System Overview

This Chapter

This chapter provides a basic description of a typical server's architecture, describing the relationships of the processors, caches, main memory, host/PCI bridges, PCI masters and targets, and E/ISA masters and targets. This subject is covered first to provide a backdrop for the subjects covered in the remainder of the book.

The Next Chapter

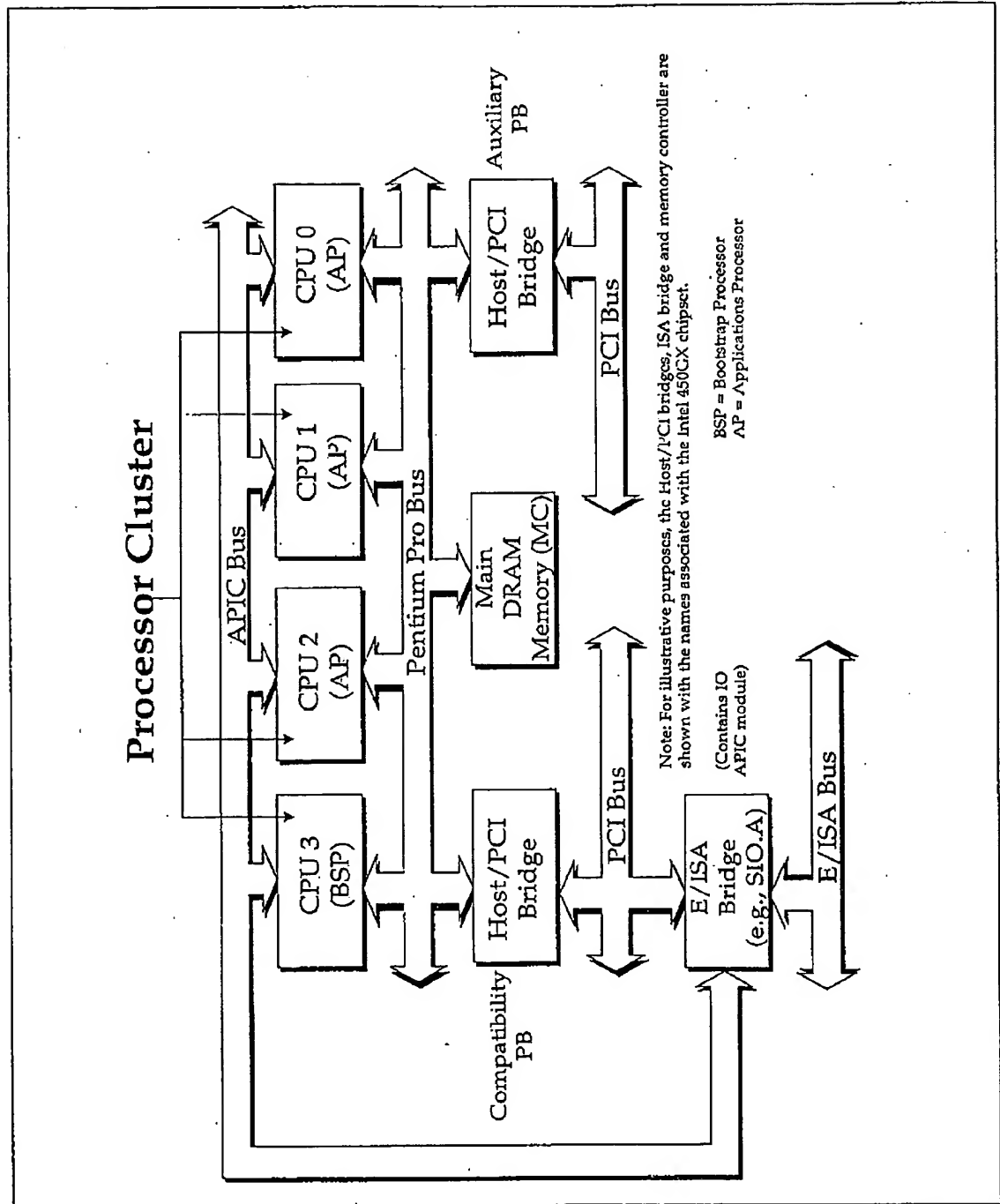
The next chapter provides a brief introduction to the internal architecture of the processor.

Introduction

Rather than launching into a detailed discussion of the Pentium Pro processor, it would be useful to provide some background regarding the processor's relationship to the other devices in the system. Figure 1-1 on page 12 illustrates a typical PC server based on the Pentium Pro processor and a typical Pentium Pro chipset. As noted in the figure, the major components have been labelled with the nomenclature of the Intel 450GX chipset. An overview of this chipset as well as the 450KX and 440FX may be found in the chapters entitled "450GX and KX Chipsets" on page 461 and "440FX Chipset" on page 497.

Pentium Pro Processor System Architecture

Figure 1-1: Block Diagram of a Typical Server



Chapter 1: System Overview

What Is a Cluster?

The term cluster refers to the processors that reside on the Pentium Pro processor bus. It can consist of anywhere from one to four processors.

What Is a Quad or 4-Way System?

A Pentium Pro system with four processors on the processor bus is frequently referred to as a quad, or 4-way system.

Bootstrap Processor

Which processor(s) begins fetching and executing the power-on self-test (POST) when reset is removed from the processors? It should be fairly obvious that all of them aren't going to execute the POST. In fact, one processor will execute the POST, configure the system board devices and enable them, and perform the boot to read the OS into memory and pass control to it.

This processor is referred to as the bootstrap processor, or BSP. The processors negotiate amongst themselves before the first instruction is fetched from memory to determine which will be the BSP. This negotiation is not performed on the host processor bus. Rather, it is performed over the APIC (Advanced Programmable Interrupt Controller) bus. This negotiation process is described in the chapter entitled "Processor Startup" on page 51.

Starting Up Other Processors

Once the BSP loads the OS and starts executing the OS kernel, the OS is responsible for detecting the presence of the additional processors and, if they are present, for initiating their program execution. In order to do this, it must be an SMP (Symmetrical Multi-Processing) OS. If it isn't, the other processors remain dormant (in other words, they're useless).

Assuming that it is an SMP OS, it assigns a task to a processor in the following manner:

- Typically, it instructs a bus mastering disk controller to load a task into memory.

Pentium Pro Processor System Architecture

- Once the task is in memory, the OS kernel, executing on the BSP, causes the BSP's internal APIC to issue a Startup IPI (Inter-Processor Interrupt, or message) to one of the other processors over the APIC bus. Supplied in this message packet is the start address of the program just placed in memory.
- Upon receipt of the Startup IPI, the target processor leaves the dormant state and begins to fetch and execute the task in memory.

Relationship of Processors to Main Memory

Each of the processors contains an L2 cache as well as L1 data and code caches. Once software enables a processor's caches and tells the processor which areas of memory it's safe to cache from (via the processor's Memory Type and Range Registers, or MTRRs), the caches begin to make copies of information from main memory (the processors are typically permitted to cache information from main memory, but not from memory on the PCI and other buses). When making a copy of memory information in the cache, the processor always copies 32-byte blocks into the cache. Each block is referred to as a line.

Information in the processor's L2 cache and L1 data cache may be stored in four possible states (referred to as the MESI cache protocol):

- **I state.** Indicates that the line is invalid (i.e., the cache doesn't have a copy).
- **E state.** Indicates that this processor's cache has a copy of the line, that it is still the same as the copy in memory, and that no other processor's cache has a copy of the same data.
- **S state.** Indicates that this processor's cache has a copy of the line, that it is still the same as memory, and that at least one other processor's cache has a copy of the same data (and it is also still the same as the copy in memory).
- **M state.** Indicates that this processor has a copy of the line, that no other processor has a copy in its cache, and that one or more of the bytes within this copy have been updated by writes from the processor core since it was read from memory. The copy in memory is stale (i.e., out-of-date).

Processors' Relationships to Each Other

As stated in the previous section, the processors read lines of information from main memory into their caches. After getting a line of data into the cache, the programmer may perform memory writes to update bytes within the line. The line is then different than memory. If the MTRRs designate the area of memory as a write-back area (covered later), the processor absorbs the new bytes into the line and marks the line modified. The update is not performed on the bus, however, so the line in memory is not updated.

Chapter 1: System Overview

When a processor performs a memory read or write transaction on the processor bus, the other processors must snoop the address in their caches and report the state of their copies to the initiator of the transaction. The initiator must do the right thing based on the snoop result. As an example, if other caches report that they have a copy of the line in the E or S state, the processor reading the line from memory must place the new line in its cache in the S state. Likewise, if a snooper had a copy in the E state, it would have to change its state from E to S as a result of watching the other processor reading the same line. Table 1-1 on page 15 contains some additional examples.

Table 1-1: Example Cache State Transitions

Stimulus	Effect on Initiator	Effect on Snoopers
Memory read initiated due to a cache miss that misses all other caches.	When read is complete, store line in E state.	None.
Memory read initiated due to a cache miss that hits on an E line in one other cache.	When read is complete, store line in S state.	The cache with the E copy must transition the state of its copy from E to S.
Memory read initiated due to a cache miss that hits on an S line in two or more other caches.	When read is complete, store line in S state.	None.
Memory read initiated due to a cache miss that hits on an M line in one other cache.	Requestor places line in S state.	Modified line is supplied directly from snooper to requestor. Memory is also updated with the fresh line. Snooper changes state of its copy from M to S.

Pentium Pro Processor System Architecture

Table 1-1: Example Cache State Transitions (Continued)

Stimulus	Effect on Initiator	Effect on Snoopers
Memory write of less than 32 bytes initiated due to a cache hit (cache line is updated) or miss in an area of memory defined as write-through memory by the MTRRs. Misses all other caches. Memory is updated.	If cache hit in initiating processor, copy is updated.	None.
Memory write of less than 32 bytes initiated due to a cache hit (cache line is updated) or miss in an area of memory defined as write-through memory by the MTRRs. Hits on an S copy in one or more caches.	If cache hit in initiating processor, copy is updated.	Snoopers transition the state of their copies from S to I (because they can't snarf the write data as it flies by on its way to memory).
Memory read and invalidate of 0 bytes initiated due to a store hit on a line in the S state in a write-back area of memory. Hits on S copy in one or more caches.	Initiator then stores into its copy and transitions it from S to M.	Snoopers transition state of their copies to I state.
Memory read and invalidate of 32 bytes initiated due to a store miss in a write-back area of memory. Hits on E copy in one other cache or an S copy in two or more.	Initiator then stores into its copy and transitions it from I to M.	Snoopers transition state of their copies to I state.

Chapter 1: System Overview

Table 1-1: Example Cache State Transitions (Continued)

Stimulus	Effect on Initiator	Effect on Snoopers
Memory read and invalidate of 32 bytes initiated due to a store miss in a write-back area of memory. Hits on M copy in one other cache.	Upon receipt of the line from the snoopers' cache, requestor immediately stores into line and transitions from I to M. Memory controller accepts line as its written from snoopers. Alternately, realizing that it's a read and invalidate, the memory controller may choose to accept the data written by the snoopers, but not actually waste time writing it into memory.	Snooper supplies requested line directly from its cache and then invalidates its copy.

Host/PCI Bridges

Bridges' Relationship to Processors

The host/PCI bridge is a bridge between the host and a PCI bus. When a processor initiates a transaction, the bridge must handle the scenarios defined in Table 1-2 on page 17.

Table 1-2: Processor-Initiated Transactions

Processor Action	Bridge Action
Processor is performing a memory read or write that targets main memory.	Ignore.

Pentium Pro Processor System Architecture

Table 1-2: Processor-Initiated Transactions (Continued)

Processor Action	Bridge Action
Processor is performing a memory read or write with a PCI memory target.	If the processor is targeting a PCI (or ISA or EISA) memory target that resides behind the bridge, the bridge must act as the surrogate target of the processor's transaction. To do this, bridge must know the memory ranges associated with the entire community of PCI (and ISA or EISA) memory targets that reside behind it (including those that may reside behind PCI-to-PCI bridges). It arbitrates for ownership of the PCI bus and reinitiates the transaction on the PCI bus. If it's a read transaction, it must return the data to the requesting processor when it receives it from the target.
Processor accesses PCI IO target.	Same actions as for processor-initiated memory access to PCI memory target. Bridge must know the IO ranges associated with the entire community of PCI (and EISA or ISA) IO targets that reside behind it (including those that may reside behind PCI-to-PCI bridges).
Processor accesses PCI device configuration register.	Bridge must compare target bus to the range of PCI buses that exists beyond the bridge. If the target bus is directly behind the bridge, initiate a type 0 PCI configuration transaction. If the target bus is beyond the bridge but isn't the one directly behind it, initiate a type 1 PCI configuration transaction. If the target bus isn't within the range of buses beyond the bridge, ignore the transaction. If the processor is targeting the bridge's configuration registers, act as the target but do not pass the transaction onto the PCI bus.
Processor initiates an interrupt acknowledge transaction.	If the 8259A interrupt controller is behind the bridge, initiate a PCI interrupt acknowledge transaction to obtain the interrupt vector from the controller.
Processor initiates a special transaction.	If the message encoded on the processor's byte enables is one that the PCI devices must receive, the bridge must initiate a PCI special cycle transaction to pass the processor's message to the PCI bus. If the message is one that the bridge itself must respond to, the bridge acts as the target of the transaction, but doesn't pass it to the PCI bus.

Chapter 1: System Overview

Bridges' Relationship to PCI Masters and Main Memory

When a PCI master initiates a transaction on the PCI bus directly behind the bridge, the bridge must act as indicated in Table 1-3 on page 19.

Table 1-3: PCI Master-Initiated Transactions

PCI Master Action	Bridge Action
PCI master initiates a memory read or write.	If the transaction targets main memory, arbitrate for ownership of the processor bus and access main memory. If a read, return the requested data to the master. The bridge must know the address range of main memory.
PCI master initiates an IO read or write.	The action taken by the bridge is bridge-specific. If there are no IO ports behind the bridge that are accessible by PCI masters, ignore (e.g., the Intel 450GX/KX chipset). If the bridge permits PCI masters to pass IO transactions onto the processor bus, the bridge arbitrates for the processor bus and passes the transaction through.
PCI master initiates a dual-address command.	If the transaction targets main memory, arbitrate for ownership of the host bus and access main memory. If a read, return the requested data to the master. The bridge must know the address range of main memory.

Bridges' Relationship to PCI Targets

The bridge must know the memory and IO ranges associated with entire community of PCI memory and IO targets that reside behind the bridge, including those that reside behind PCI-to-PCI bridges.

Bridges' Relationship to EISA or ISA Targets

The bridge must know if the EISA or ISA bus is located behind it. If it is, the bridge must act as the target when a processor transaction targets a memory or IO address that may be for an EISA or ISA target.

Pentium Pro Processor System Architecture

Bridges' Relationship to Each Other

If there are two host/PCI bridges, they both share access to the BPRI# signal to request host bus ownership. Only one device is permitted to use this signal at a time. If the two bridges both require access to main memory, they must arbitrate amongst themselves for ownership of the BPRI# signal. In the 450GX chipset, two sideband signals, IOREQ# and IOGNT#, are used for this purpose. The compatibility PB (i.e., the bridge with the ISA or EISA bridge behind it) has higher priority than the aux PB (when both bridges simultaneously require access to main memory).

Bridge's Relationship to EISA and ISA Masters and DMA

When an EISA master, an ISA master, or a DMA channel in the EISA or ISA bridge requires access to main memory, the EISA or ISA bridge initiates a memory access on the PCI bus. The host/PCI bridge between the EISA or ISA bridge and main memory must arbitrate for ownership of the processor bus and access main memory. Because EISA and ISA masters and DMA channels are frequently sensitive to memory access time, the host/bridge between them and main memory is typically assigned higher processor bus priority than the other host/PCI bridge. This feature is typically referred to as GAT (Guaranteed Access Time).

Part 2: Processor's Hardware Characteristics

The Previous Part

Part 1 provided a basic description of a typical server's architecture, describing the relationships of the processors, caches, main memory, host/PCI bridges, PCI masters and targets, and E/ISA masters and targets. This subject was covered first to provide a backdrop for the subjects covered in the remainder of the book.

This Part

Part 2 provides a description of the processor's internal and external hardware characteristics. It is divided into the following sections, each consisting of one or more chapters:

- "Hardware Section 1: The Processor" on page 23.
- "Hardware Section 2: Bus Intro and Arbitration" on page 177.
- "Hardware Section 3: The Transaction Phases" on page 239.

The Next Part

Part 3 provides a description of the Pentium Pro processor's enhancements to the software environment.

Hardware

Section 1:

The Processor

This Section

Part 1, Section 1 focuses on the processor's internal operation. The chapters that comprise this section are:

- "Processor Overview" on page 25.
- "Processor Power-On Configuration" on page 37.
- "Processor Startup" on page 51.
- "The Fetch, Decode, Execute Engine" on page 61.
- "Rules of Conduct" on page 119.
- "The Processor Caches" on page 133.

The Next Section

The chapters that comprise Part 1, Section 2 introduce the processor's bus and transaction protocol.

Warning

The majority of the processor's signal pins are active low signals (e.g., all of the pins involved in a transaction). All tables in the *Intel Pentium Pro Volume One* data book, however, represent an asserted signal (in other words, in the electrically low state) with a one, while deasserted signals (electrically high) are represented by a zero in table entries. In other words, a "logical" one in a table indicates that the respective signal pin is asserted (in the electrically low state).

As an example, when a table entry indicates a one for the HITM# signal state, this indicates that it is asserted (electrically low).

2

Processor Overview

The Previous Chapter

The previous chapter provided a basic description of a typical server's architecture, describing the relationships of the processors, caches, main memory, host/PCI bridges, PCI masters and targets, and E/ISA masters and targets. This subject was covered first to provide a backdrop for the subjects covered in the remainder of the book.

This Chapter

This chapter provides a brief introduction to the internal architecture of the processor.

The Next Chapter

The next chapter describes the manner in which the processor automatically configures some of its operational characteristics at power up time upon the removal of reset. In addition, it describes some of the basic processor features that may be enabled or disabled by the programmer.

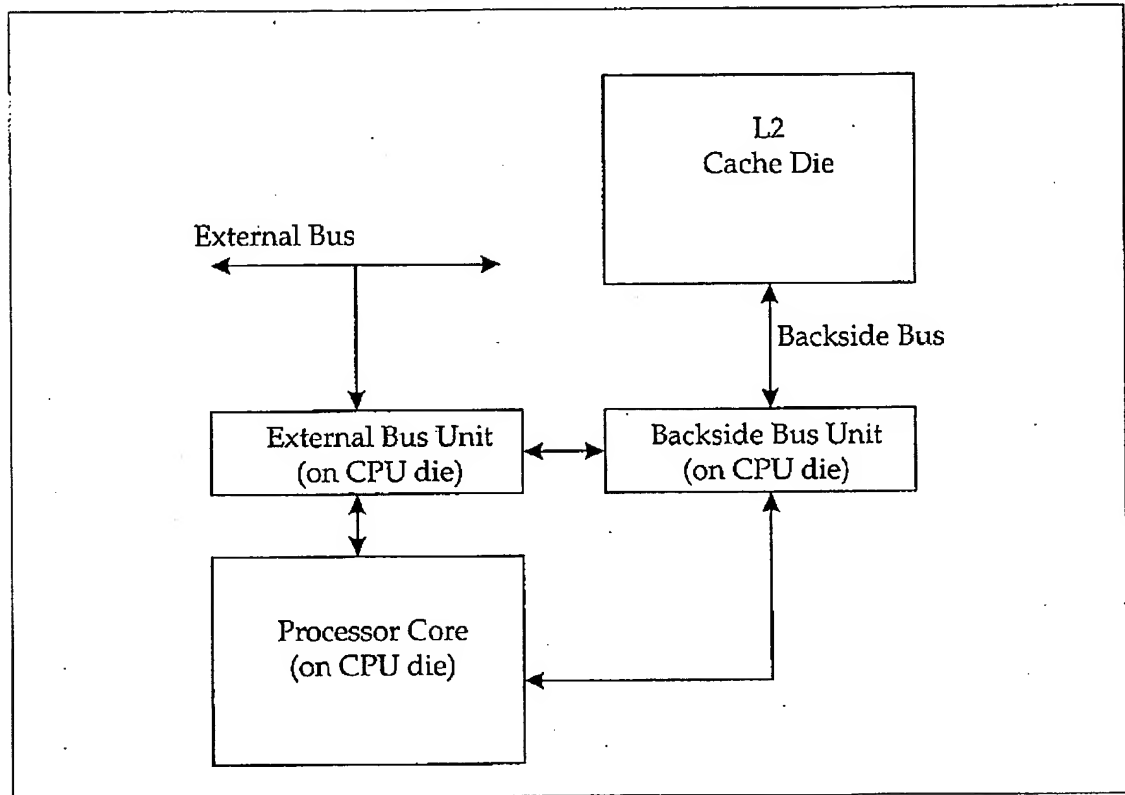
Two Bus Interfaces

Refer to Figure 2-1 on page 26. The current implementations of the processor package consist of two tightly-coupled dies (set in a dual-cavity package), one containing the processor itself, while the other contains the L2 cache. The processor is connected to the L2 die over a dedicated bus (sometimes referred to as the backside bus) consisting of a 36-bit address bus and a 64-bit data bus. In addition, the processor's bus interface unit is connected to the external world via the external bus. In the event of a miss on the L1 data or code cache, the L2 cache can be accessed via the backside bus at the same time that the processor

Pentium Pro Processor System Architecture

(or an another bus agent) is using the external bus. Future versions of the processor (e.g., Klamath) may eliminate the L2 die from the package. In this case, the backside bus interface pins will be made available on the processor package, permitting the system board designer to connect the processor to an external L2 cache.

Figure 2-1: Two Bus Interfaces



External Bus

Bus on Earlier Processors Inefficient for Multiprocessing

The processor's external bus is significantly different than that found on the earlier processors. The older buses were designed to be used by one initiator at a time. Each transaction consisted of an address and a data phase. When a proces-

Chapter 2: Processor Overview

processor gained ownership of the bus, it owned the entire bus and no other processor could use the bus until the transaction in progress completed.

Pentium Bus has Limited Transaction Pipelining Capability

It should be noted that a small degree of transaction overlap is possible on the Pentium bus (see Figure 2-2 on page 28). If the target of the transaction (e.g., the external L2 cache) supports transaction pipelining, it asserts NA# (Next Access) to the processor, thereby granting it permission to initiate another transaction before the data transfer for the current transaction is completed. In a dual-processor Pentium configuration, either the same processor that initiated the first transaction, or the other processor in the pair can then initiate the address phase of a new transaction. However, the processor that initiated the second transaction (or the target, if a read) cannot transfer data over the data bus until the data transfer for the first transaction completes. The Pentium processor can only keep track of up to two currently-outstanding (i.e., pipelined) transactions.

Many target devices are extremely slow to provide read data or to accept write data. A classic example is a host/PCI bridge. Before it can supply a processor with read data from a PCI device, it must first arbitrate for PCI bus ownership. This can take quite a while if the PCI bus is currently in use by another PCI master. In addition, other masters may also be requesting bus ownership and it may be their turn rather than the host/PCI bridge's. Once ownership is acquired, the bridge then must wait for the target device to supply the read data. It's possible that the read may have to traverse a PCI-to-PCI bridge to get to the target. The net result can be a very lengthy delay before the read data is presented to the processor that requested. The Pentium's data bus would be tied up until the data is presented and BRDY# is asserted. Only then can the processor read the data and end the transaction.

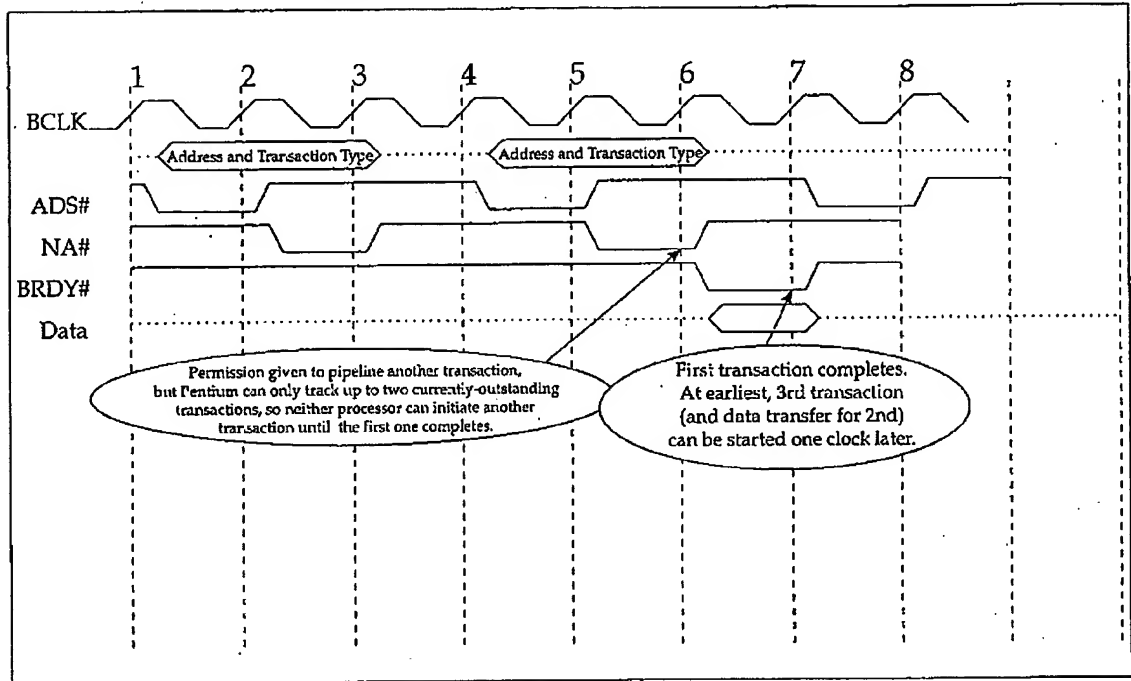
The Pentium processor and bus have the following drawbacks:

- Processor can only keep track of up to two transactions currently-outstanding on the bus.
- Targets have no way of issuing a retry to a processor if they cannot handle a transaction immediately. The bus remains busy until the data transfer finally takes place.
- A very slow access target cannot memorize a transaction, disconnect from the initiator, process the transaction off-line, and reconnect with the initiator when finally ready to transfer data. This is called transaction deferral.

Pentium Pro Processor System Architecture

This approach is fine for a single-processor environment, but terrible for an environment that incorporates multiple processors on the host bus.

Figure 2-2: Pentium Transaction Pipelining



Pentium Pro Bus Tuned for Multiprocessing

The Pentium Pro bus protocol prevents bus stalls in three basic ways:

- In a typical Pentium Pro bus environment, up to eight transactions can be currently outstanding at various stages of completion.
- If the target of the transaction cannot deal with the transaction right now, rather than tie up the bus by inserting wait states, it will issue a retry to the initiator. This causes the initiator to wait a little while and try the transaction again at a later time. This frees up the bus for other initiators.
- If the target of a read or write transaction realizes that it will take a fairly long time to complete the data transfer (i.e., provide read data or accept write data), it can tell the initiator to break the connection and that the target will initiate a transaction later to complete the transaction. This is referred to as transaction deferral.

Chapter 2: Processor Overview

These mechanisms prevent any properly-designed bus agent from tying up the bus for extended periods of time. A detailed description of the processor's bus is presented later in the book.

IA = Legacy

With the advent of the Pentium Pro, Intel refers to the x86 instruction and register sets as the Intel Architecture ("IA" for short) instruction and register sets. In essence, IA refers to the legacy architecture inherited by the designers of the Pentium Pro and subsequent generations of the processor family. The instructions are complex, requiring a number of resources for execution, and the register set is quite small.

Instruction Set

IA Instructions Vary in Length and are Complex

Many IA instructions are complex in nature, requiring extensive decode and execution logic. Many of them require a number of execution resources in order to complete. As an example, one IA instruction may entail one or more memory reads, one or more computations, and one or more memory writes. In addition, an IA instruction may be anywhere from one to 15 bytes in length. This presents the processor designer with a formidable obstacle with regard to dispatching and executing more than one instruction simultaneously (i.e., designing a superscalar processor). The processor has to pre-scan the instruction stream (i.e., the prefetched instructions) in order to determine where one instruction ends and another begins. Only then could the processor logic decode the instructions and determine whether or not they can be executed simultaneously. The instructions that comprise a sequence of IA instructions are frequently very co-dependent on each other, thereby making it very difficult to execute the instructions in parallel with each other.

Pentium Pro Translates IA Instructions into RISC Instructions

Instructions are prefetched from memory and are placed into the L2 cache and the L1 code cache. The code cache feeds the instruction pipeline in the processor

Pentium Pro Processor System Architecture

core. As blocks of instructions are obtained from the cache, the processor core parses them to identify the boundaries between instructions. It then decodes the variable-length IA instructions into fixed length RISC instructions referred to as micro-ops (or uops). The micro-ops are then passed along to an instruction pool where they await dispatch and execution. The dispatch logic can see all 40 entries in the pool and will dispatch any instruction for execution if the appropriate execution unit and the data operands it requires are available.

In-Order Front End

Instructions are prefetched, decoded and placed into the instruction pool in strict program order (i.e., the order in which the programmer expects them to be executed).

Out-of-Order Middle

Once the micro-ops are placed into the instruction pool, however, they can be executed out-of-order (i.e., not in the original program order). In other words, the dispatch logic does not have to wait for all instructions that precede an instruction to complete execution before executing the instruction. It will dispatch any instruction for execution if the appropriate execution unit and the data operands the instruction requires are available. It should be noted that a copy of the micro-op remains in the pool until the instruction is retired (explained in the text that follows). The results of the speculatively-executed instruction are stored in the instruction's pool entry rather than in the processor's register set, however.

The instruction may have been executed before a conditional branch that resides earlier in the program flow. Only when it is certain that all upstream branches have been resolved and that the instruction should have been executed, is it marked as ready for retirement. If it turns out that the instruction should not have been executed, the micro-op and its results are discarded from the pool.

In-Order Rear End

In order to have the program appear to execute in the order intended by the programmer, the processor core retires the instructions from the pool and commits their results from the pool locations to the processor's real register set in original program order.

Chapter 2: Processor Overview

A detailed description of the processor core can be found in the chapter entitled "The Fetch, Decode, Execute Engine" on page 61.

Register Set

IA Register Set is Small

The IA register set implemented in earlier x86 processors is extremely small. The small number of registers permits the processor (and the programmer) to keep only a small number of data operands close to the execution units where they can access them quickly. Rather, the programmer is frequently forced to write back the contents of one or more of the processor's registers to memory when he or she needs to read additional data operands from memory to be operated on. Later, when the programmer requires access to the original set of data operands, they must again be read from memory (perhaps after first storing the current contents of the registers). This juggling of data between the register set and memory takes time and exacts a penalty (perhaps severe) on the performance of the program. Figure 2-3 on page 32 illustrates the IA register set.

Pentium Pro Processor System Architecture

Figure 2-3: IA General Register Set

	31	23	15	8	7	0
EAX				AH	AX	AL
EBX				BH	BX	BL
ECX				CH	CX	CL
EDX				DH	DX	DL
EBP				BP		
ESI				SI		
EDI				DI		
ESP				SP		

Pentium Pro has 40 General-Purpose Registers

Rather than the extremely limited register set pictured in Figure 2-3 on page 32, the Pentium Pro has 40 registers. As described earlier, IA instructions are translated into fixed-length micro-ops prior to being executed. When executed, the micro-op may:

- have to place a value into one of the IA general registers.
- have to read a value (from an IA register) that was placed into it by an instruction that was executed earlier.
- when executed, change the contents of the EFLAGS or FPU status register.

Remember that the processor core permits instructions to be executed out-of-order (referred to as speculative execution). Imagine what might happen if the results of the instruction's execution were immediately committed to the processor's register set. Values in registers would be changed and condition bits in the EFLAGS and FPU status registers would be updated, perhaps erroneously and certainly not in the expected order.

Chapter 2: Processor Overview

Rather than immediate commitment of instruction results to the real register set, the processor stores the result of an instruction's execution in the instruction's pool entry. If, when executed, another micro-op requires the result produced by micro-ops that precede it in program flow, the result(s) are forwarded to it directly from the pool entries of the other micro-ops (assuming that they have completed execution). If a micro-op has been dispatched for execution and another micro-op that requires the results of its execution is queued for dispatch to an execution unit, the result is forwarded directly from the execution unit to the queued micro-op (and is also stored in the pool entry associated with the micro-op that produced the result). This is referred to as *feed forwarding*.

Rerouting accesses intended for the IA register set to the larger pool register set is necessary for speculative execution and is referred to as register aliasing.

Elimination of False Register Dependencies

Consider the following example:

```
mov  eax,17      ;17 -> eax
add  mem,eax      ;memory loc.= eax + content of memory loc.
mov  eax,3        ;3. -> eax
add  eax,ebx      ;eax = ebx + eax
```

In earlier x86 processors, these instructions would have to be executed one at a time in order to yield the correct results. The processor couldn't execute the third and fourth instructions before the 1st and 2nd had completed (because the 2nd instruction must use the value placed into `eax` before the 3rd instruction can place a new value into `eax`).

The Pentium Pro processor recognizes that the `eax` register needn't be loaded with the value 17. Rather, the same result can be produced by just adding 17 to the memory location. Likewise, `eax` doesn't need to be loaded with the value 3. Instead, the value 3 can just be added to `ebx` and the result stored in `eax`. The processor can execute instructions 1 and 3 simultaneously, placing the values 17 and 3 into the pool entries for these two micro-ops. Likewise, instructions 2 and 4 can then be executed in parallel, obtaining the values 17 and 3 from the pool entries associated with micro-ops 1 and 3.

Pentium Pro Processor System Architecture

Introduction to the Internal Architecture

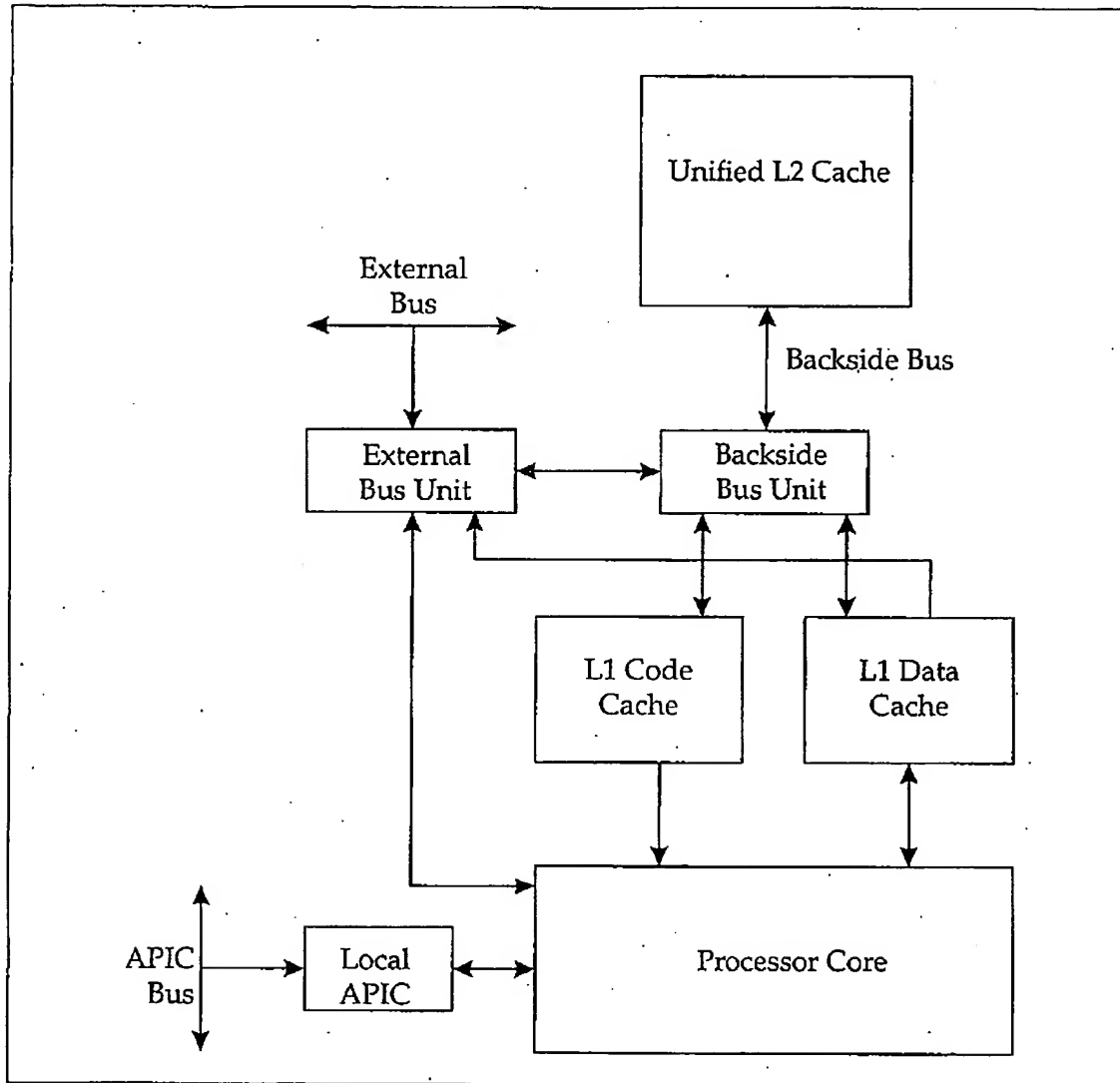
Refer to Figure 2-4 on page 35. The Pentium Pro processor consists of the following basic elements:

- **External bus unit.** Performs bus transactions when requested to do so by the L2 cache or the processor core.
- **Backside bus unit.** Interfaces the processor core to the unified L2 cache.
- **Unified L2 cache.** Services misses on the L1 data and code caches. When necessary, issues requests to the external bus unit.
- **L1 data cache.** Services data load and store requests issued by the load and store execution units. When a miss occurs, forwards request to the L2 cache.
- **L1 code cache.** Services instruction fetch requests issued by the instruction prefetcher.
- **Processor core.** The processor logic responsible for the following:
 - Instruction fetch.
 - Branch prediction.
 - Parsing of IA instruction stream.
 - Decoding of IA instructions into RISC instructions (referred to as micro-ops, or uops).
 - Mapping accesses for IA register set to a larger physical register set.
 - Dispatch, execution and retirement of micro-ops.
- **Local APIC unit.** Responsible for receiving interrupt request from other processors, the processor local interrupt pins, the APIC timer, APIC error conditions, performance monitor logic, and the IO APIC module. These requests are then prioritized and forwarded to the processor core for execution.

A detailed description of the processor's internal operation can be found in subsequent chapters.

Chapter 2: Processor Overview

Figure 2-4: Simplified Processor Block Diagram



3

Processor Power-On Configuration

The Previous Chapter

The previous chapter provided a brief introduction to the internal architecture of the processor.

This Chapter

This chapter describes the manner in which the processor automatically configures some of its operational characteristics at power up time upon the removal of reset. In addition, it describes some of the basic processor features that may be enabled or disabled by the programmer.

The Next Chapter

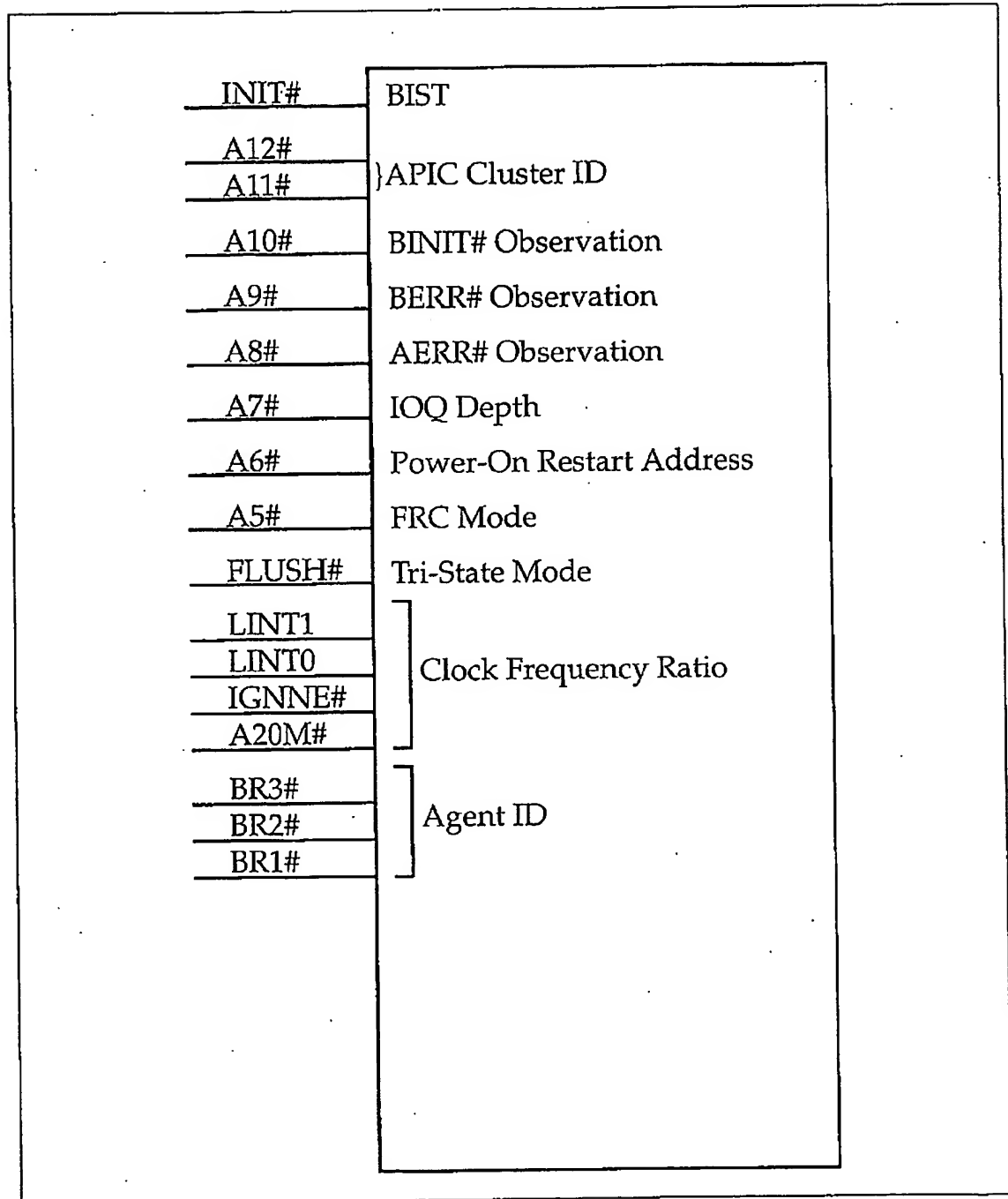
The next chapter describes the processor state at startup time. It also describes the process that the processors engage in to select the processor that will fetch and execute the power-on self-test (POST), as well as how, after it has been loaded, an SMP OS can detect the additional processors and assign tasks for them to execute.

Features that are Automatically Configured

The processor samples a subset of its signal pins on the trailing-edge of reset to configure certain of its operational characteristics. Figure 3-1 on page 38 illustrates the signals that are sampled and the features associated with each. The sections that follow describe each of these selections and the pins sampled to determine the selection.

Pentium Pro Processor System Architecture

Figure 3-1: Automatic Feature Determination



Chapter 3: Processor Power-On Configuration

Setup and Hold Time Requirements

In order to reliably sample the values presented on its pins on the trailing-edge of the reset signal, the following setup and hold times must be met:

- the signal must be in the appropriate state for at least four bus clocks before the trailing-edge of RESET#. This is the setup time requirement.
- signal must be held in that state for at least two but not greater than 20 bus clocks after RESET# is deasserted. This is the hold time requirement.

Run BIST Option

If the INIT# pin is sampled in the low state at the trailing-edge of RESET#, the processor will execute its internal Built-In Self-Test prior to initiation of program fetch and execution. This takes approximately 5.5 million processor clock cycles to complete on the Pentium Pro processor, but it should be noted that the exact clock count is processor-specific and can change. If the BIST completes successfully, EAX contains zero. If an error is incurred during the BIST, however, EAX contains a non-zero error code. Intel does not provide a breakdown of the error codes. When the BIST is not invoked, EAX contains zero when program execution is initiated after reset's removal. In either case, the programmer should check to ensure that EAX is clear at the start of the Power-On Self-Test and not proceed with the POST if it contains a non-zero value.

Error Observation Options

The processor (actually, any bus agent) may be configured to monitor or to ignore three of its error inputs that are used to signal errors during bus transactions. Those three inputs are:

- **AERR# (Address Error).** Sampling A8# low at the trailing-edge of reset configures the processor to sample AERR# at the end of the error phase of transactions initiated by itself or by other bus agents. Sampling A8# high configures the processor to ignore AERR#. This is referred to as the processor's *AERR# observation policy*. A detailed discussion of AERR# may be found in the section entitled "Error Phase" on page 253.
- **BERR# (Bus Error).** Sampling A9# low at the trailing-edge of reset configures the processor to monitor BERR#. Sampling A9# high configures the processor to ignore BERR#. This is referred to as the processor's *BERR#*

Pentium Pro Processor System Architecture

observation policy. A detailed discussion of BERR# may be found in the chapter entitled "Other Signals" on page 345. It should be noted that, although other bus agents may be configured to do so, the Pentium Pro processor does not support BERR# sampling.

- **BINIT#** (Bus Initialization). Sampling A10# low at the trailing-edge of reset configures the processor to sample BINIT#. Sampling A10# high configures the processor to ignore BINIT#. This is referred to as the processor's *BINIT# observation policy*. A detailed discussion of BINIT# may be found in the chapter entitled "Other Signals" on page 345.

In-Order Queue Depth Selection

If the processor samples its A7# pin low at the trailing-edge of reset, it configures the depth of its In-Order Queue (IOQ) to one. Sampling A7# high configures its queue depth to eight. The IOQ is described in subsequent chapters.

Power-On Restart Address Selection

Sampling A6# low configures the processor to initiate program execution at memory address 0000FFFF0h (the PC-compatible POST entry point 16 locations from the top of the first MB).

Sampling A6# high configures the processor to initiate program execution at memory address 0FFFFFFF0h (16 locations from the top of the first 4GB).

FRC Mode Enable/Disable

Sampling A5# low configures a processor pair to act as a Functional Redundant Checker (FRC) pair. A detailed discussion of FRC mode can be found in "Processor's Agent and APIC ID Assignment" on page 42.

Sampling A5# high configures the processor to act normally after reset is removed (i.e., begin program execution).

APIC ID Selection

Each processor contains a local APIC module. This module must be assigned two addresses at startup time:

Chapter 3: Processor Power-On Configuration

- **Cluster ID.** Identifies what cluster of processors the processor (and therefore its encapsulated APIC) belong to. The processor may be assigned a cluster number of 0, 1, 2, or 3. A[12:11]# are sampled at the trailing-edge of reset to determine the cluster ID (the following binary values are sampled electrical values): 00b = 3, 01b = 2, 10b = 1, and 11b = 0.
- **APIC ID.** Identifies the number of the processor (and therefore that of its encapsulated APIC) within its cluster. At the trailing-edge of reset, the processor is assigned a processor number of 0, 1, 2, or 3 when it samples the value present on its BR[3:1]# inputs. A description of this process can be found in this chapter under the heading "Processor's Agent and APIC ID Assignment" on page 42.

Selecting Tri-State Mode

In a test environment, the processor can be configured to disconnect from (i.e., to tri-state its output drivers) all of its output pins. This permits an external board tester to drive all of the signal traces (normally driven by the processor) to known values. Using a bed-of-nails fixture that permits probing of all system board nodes, the board tester can then verify that all of these signal traces are properly connected to all of the appropriate nodes on the board.

The processor is configured to tri-state all of its output drivers when it samples a low on its FLUSH# pin at the trailing-edge of reset.

Processor Core Speed Selection

The processor's bus clock (BCLK) is supplied directly to its bus interface by an external clock connected to its BCLK input pin. The processor's internal clock is supplied by an internal phase-locked loop (PLL) that multiplies the BCLK by a value determined at startup time to yield the internal processor clock (PCLK).

The processor determines the PLL multiplication factor by sampling the values supplied on its A20M#, IGNNE#, LINT1, and LINT0 pins during reset. Specifically, it starts sampling these pins when reset is first asserted and continues to sample them throughout the reset assertion period. It latches the value on the trailing-edge of reset and uses the latched value as the PLL multiplication factor for the entire power up session. Table 3-1 on page 42 defines the available selections (note that all multipliers are not supported by all processor variants).

The setup time for these four pins must be at least 1ms prior to reset deassertion and the values presented must remain stable throughout this period and for at

Pentium Pro Processor System Architecture

least two additional BCLKs after reset is removed (in other words, the hold time for them is 2 BCLKs).

Table 3-1: Core Speed Selection

Multiplier	LINT1	LINT0	IGNNE#	A20M#
2	0	0	0	0
3	0	0	1	0
4	0	0	0	1
5/2	0	1	0	0
7/2	0	1	1	0
Reserved	0011b, 0101b, 0111b - 1110b			
2	1	1	1	1

Processor's Agent and APIC ID Assignment

Each of the processors in the cluster is assigned an agent ID at power up time. This value is also loaded into the processor's APIC ID register. When a processor needs the bus to initiate a transaction, it must first request bus ownership. The processor's agent ID is used during the bus arbitration process (a detailed description of processor bus arbitration may be found in the chapter entitled "Obtaining Bus Ownership" on page 201). The processor's APIC ID is used as an address by the IO APIC module and the local APICs in other processors to transmit messages to a specific processor over the APIC bus.

Refer to Figure 3-2 on page 44 and Table 3-2 on page 43. A processor determines its agent and APIC IDs at power up time by sampling the state of its BR[3:1]# pins on the trailing-edge of reset. The system board designer ensures that each of the processors within the cluster is assigned a unique ID by asserting the BREQ0# signal line. BREQ0# must be asserted for at least four BCLKs prior to the trailing edge of reset and must be held asserted for an additional two BCLKs.

Each processor also samples its A5# pin on reset's trailing-edge to determine whether it is to operate as a master (i.e., normal operation) or a checker (in FRC mode). For a detailed description of FRC mode, refer to "FRC Mode" on

Chapter 3: Processor Power-On Configuration

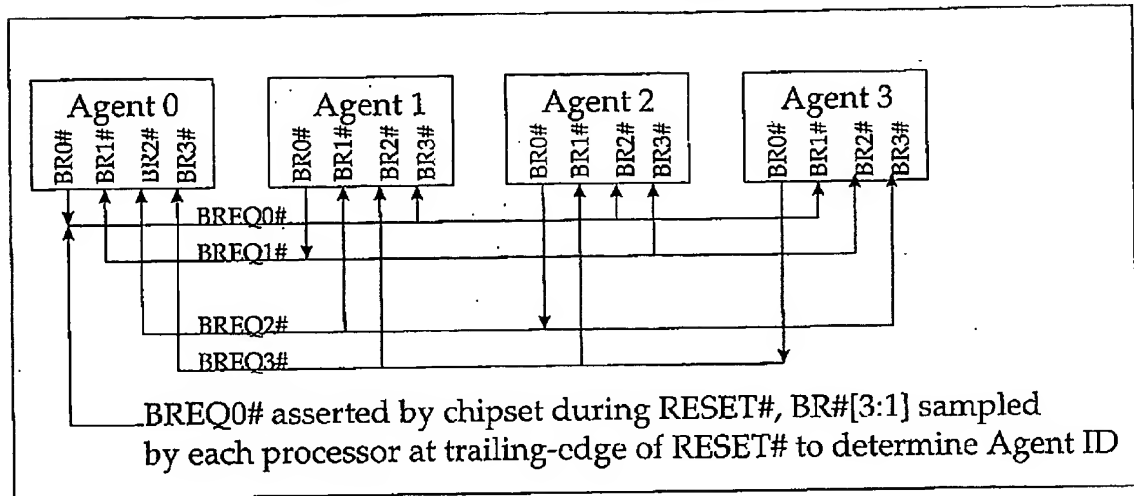
page 44. If the processor that would normally be designated as agent 1 samples a low on A5#, it is assigned an agent ID of 0 and will act as the checker for agent 0. Likewise, if the processor that would normally be designated as agent 3 samples a low on A5#, it is assigned an agent ID of 2 and will act the checker for agent 2.

Table 3-2: Processor Agent ID Assignment

BR1#	BR2#	BR3#	A5#	Agent ID
FRC mode disabled (A5# = 1)				
1	1	1	1	Agent ID = 0 and processor operates as a master.
1	1	0	1	Agent ID = 1 and processor operates as a master.
1	0	1	1	Agent ID = 2 and processor operates as a master.
0	1	1	1	Agent ID = 3 and processor operates as a master.
FRC mode enabled (A5# = 0)				
1	1	1	0	Agent ID = 0 and processor operates as a master.
1	1	0	0	Agent ID = 0 and processor operates as a checker for agent 0 (see previous row).
1	0	1	0	Agent ID = 2 and processor operates as a master.
0	1	1	0	Agent ID = 2 and processor operates as a checker for agent 2 (see previous row).

Pentium Pro Processor System Architecture

Figure 3-2: Processor Agent ID Assignment



FRC Mode

When a pair of processors (processors 0 and 1, or 2 and 3) are setup as a master/checker pair, the checker is assigned the same agent ID as the master it is paired with. From that point forward, the checker takes the following actions:

- It never drives the bus.
- Whenever it sees its partner initiate a memory code read transaction, it accepts the code read from memory at the same time that its partner does.
- It decodes the instructions that are fetched and executes them in synchronism with its partner (they are sync'd to the same clock).
- It watches the bus to see that its partner, executing the same code stream, does everything that it would do, clock-by-clock.
- If, in any BCLK, it sees its partner do anything that it wouldn't do, it asserts its FRCERR output to inform the system board logic that a miscompare has occurred.

To exit FRC mode, reset must be reasserted, the value of A5# changed, and reset is then removed.

Chapter 3: Processor Power-On Configuration

Program-Accessible Startup Features

One of the processor's MSR (model-specific registers) provides the programmer with two capabilities:

- the ability to determine the state of the features that were automatically enabled or disabled on the trailing-edge of reset (when the processor sampled a subset of its pins).
- the ability to programmatically select other processor features.

The EBL_CR_POWERON MSR (external bus logic poweron configuration register) contains a series of read-only bits used to indicate the automatic selections, as well as a series of read/write bits used to programmatically enable/disable other features. This register is accessed using the RDMSR and WRMSR instructions (described in the chapter entitled "Instruction Set Enhancements" on page 359). The EBL_CR_POWERON register is pictured in Figure 3-3 on page 50 and described in Table 3-3 on page 46.

Pentium Pro Processor System Architecture

Table 3-3: Bit Assignment of EBL_CR_POWERON MSR

Bit	Read/ Write	Description
0	R/W	<p>Data bus error code policy. Each of the processor's eight data paths includes a ninth bit that may be used either as an ECC or a parity bit.</p> <ul style="list-style-type: none"> • 1 = ECC. Enables processor to utilize its ECC algorithm when transferring one or more bytes over the data bus. The processor's ECC algorithm can detect and correct single-bit errors within a byte (referred to as SEC, or single-bit error correction); can detect (but not correct) double-bit errors within a byte (referred to as DED, or double-bit error detection); and can detect (but not correct) all errors confined to a single nibble within the byte (referred to as S4ED, or single group of 4 error detection). • 0 = Parity. With this selection, the processor uses the ninth bit associated with each of its eight data paths as a parity rather than an ECC bit. When reading a byte over a data path, the processor checks for an even number of electrical lows in the nine bit pattern. When writing a byte, it sets the ninth bit either high or low to force the overall nine bit pattern to an even number of electrical lows.
1	R/W	<p>1 = disable data bus error checking. 0 = enable data bus error checking. Also see bit 0 description.</p>
2	R/W	<p>1 = disable response bus parity checking. 0 = disable response bus parity checking. For additional information, refer to "Response Phase Signal Group" on page 278.</p>
3	R/W	<p>1 = disable processor's ability to assert AERR# upon detection of bad request information received from another initiator. 0 = enable. For additional information, refer to the section on the Error Phase in the chapter entitled "Error Phase" on page 253.</p>

Chapter 3: Processor Power-On Configuration

Table 3-3: Bit Assignment of EBL_CR_POWERON MSR (Continued)

Bit	Read/ Write	Description
4	R/W	1 = disable processor's ability to assert BERR# when it is acting as the initiator and observes an unrecoverable error. 0 = enable processor's ability to assert BERR# when it is acting as the initiator and observes an unrecoverable error. For a detailed description of BERR#, refer to the chapter entitled "Other Signals" on page 345.
5	x	Reserved
6	R/W	1 = disable processor's ability to assert BERR# (along with IERR#) when an internal error occurs. 0 = enable processor's ability to assert BERR# (along with IERR#) when an internal error occurs. For a detailed description of BERR#, refer to the chapter entitled "Other Signals" on page 345.
7	R/W	1 = disable processor's ability to assert BINIT# if the processor has lost track of the transactions currently outstanding on the bus. 0 = enable processor's ability to assert BINIT# if the processor has lost track of the transactions currently outstanding on the bus. For a detailed description of BINIT#, refer to the chapter entitled "Other Signals" on page 345.
8	R	1 = Output tri-state mode enabled. 0 = Output tri-state mode disabled. This option was automatically set on the trailing-edge of reset when FLUSH# was sampled. For more information, refer to "Selecting Tri-State Mode" on page 41.
9	R	1 = BIST enabled. 0 = BIST disabled. This option was automatically set on the trailing-edge of reset when INIT# was sampled. For more information, refer to "Run BIST Option" on page 39.

Pentium Pro Processor System Architecture

Table 3-3: Bit Assignment of EBL_CR_POWERON MSR (Continued)

Bit	Read/Write	Description
10	R	1 = AERR# observation enabled. 0 = AERR# observation disabled. This option was automatically set on the trailing-edge of reset when A8# was sampled. For more information refer to "Error Observation Options" on page 39, and to the section on the Error Phase in the chapter entitled "Error Phase" on page 253.
11	X	Reserved
12	R	1 = BINIT# observation enabled. 0 = BINIT# observation disabled. This option was automatically set on the trailing-edge of reset when A10# was sampled. For more information on BINIT#, refer to the chapter entitled "Other Signals" on page 345 and to "Error Observation Options" on page 39.
13	R	1 = IOQ depth of 1. 0 = IOQ depth of 8. This option was automatically set on the trailing-edge of reset when A7# was sampled. For more information on the IOQ depth selection, refer to "In-Order Queue Depth Selection" on page 40.
14	R	1 = Poweron restart address is 0000FFFF0h (top of 1st MB). 0 = Poweron restart address is 0FFFFFFF0h (top of 1st 4GB). This option was automatically set on the trailing-edge of reset when A6# was sampled. For more information, refer to "Power-On Restart Address Selection" on page 40.
15	R	1 = FRC mode enabled. 0 = FRC mode disabled. This option was automatically set on the trailing-edge of reset when A5# was sampled. For more information, refer to "FRC Mode Enable/Disable" on page 40.
17:16	R	APIC cluster ID. This ID was automatically set on the trailing-edge of reset when A[12:11]# were sampled. For more information, refer to "APIC ID Selection" on page 40.
19:18	X	Reserved

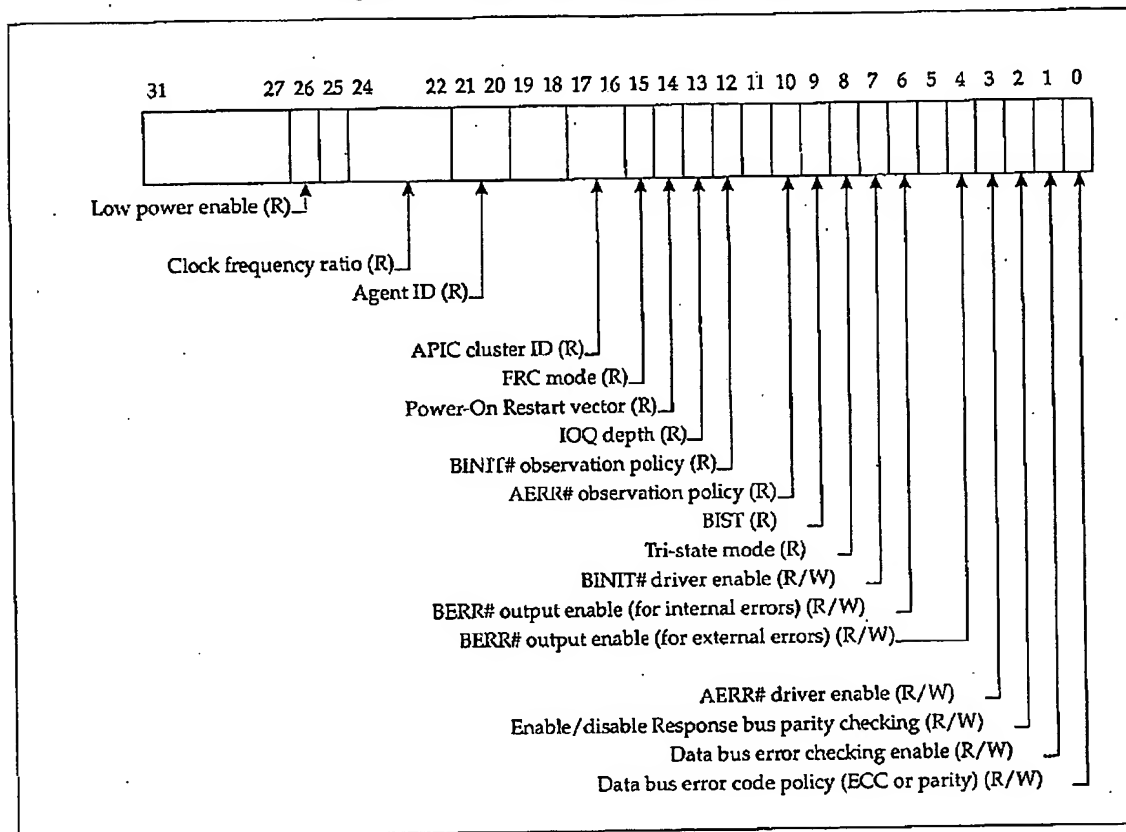
Chapter 3: Processor Power-On Configuration

Table 3-3: Bit Assignment of EBL_CR_POWERON MSR (Continued)

Bit	Read/ Write	Description
21:20	R	Symmetric arbitration ID. Also referred to as the processor's agent or CPU ID. This option was automatically set on the trailing-edge of reset when BR[3:1]# were sampled. For more information, refer to "Processor's Agent and APIC ID Assignment" on page 42, and to the chapter entitled "Obtaining Bus Ownership" on page 201.
24:22	R	Clock frequency ratio. This option was automatically set on the trailing-edge of reset when LINT[1:0]#, IGNNE# and A20M# were sampled. For more information, refer to "Processor Core Speed Selection" on page 41.
25	X	Reserved
26	R	Low power enable. The Intel documentation provides no explanation of this bit. However, the author is pretty sure that, when set to one, it indicates that the processor has entered the Stop Grant state as a result of the assertion of STPCLK# to the processor. For more information, refer to "STPCLK#" on page 355.
31:27	X	Reserved

Pentium Pro Processor System Architecture

Figure 3-3: EBL_CR_POWERON MSR



4

Processor Startup

The Previous Chapter

The previous chapter described the manner in which the processor automatically configures some of its operational characteristics at power up time upon the removal of reset. In addition, it described some of the basic processor features that may be enabled or disabled by the programmer.

This Chapter

This chapter describes the processor state at startup time. It also describes the process that the processors engage in to select the processor that will fetch and execute the power-on self-test (POST), as well as how, after it has been loaded, an SMP OS can detect the additional processors and assign tasks for them to execute.

The Next Chapter

The next chapter provides a detailed description of the processor logic responsible for instruction fetch, decode, and execution.

Selection of Processor's Agent and APIC IDs

At the trailing-edge of reset, the processor samples its BR[3:1]# inputs to determine its agent ID. The agent ID also serves as its APIC ID and as its APIC's startup arbitration ID (when arbitrating for ownership of the APIC bus). Refer to "Processor's Agent and APIC ID Assignment" on page 42 and to the chapter entitled "Obtaining Bus Ownership" on page 201 for additional information regarding processor agent ID assignment.

Pentium Pro Processor System Architecture

Processor's State After Reset

The assertion of the processor's RESET# input has the effects indicated in table Table 4-1 on page 52.

Table 4-1: Effects of Reset on CPU

Effect	Result
L2 cache	All entries in the L2 Cache are invalidated.
L1 code cache	All entries in the L1 code cache are invalidated.
L1 data cache	All entries in the L1 data cache are invalidated.
Branch Target Buffer (BTB)	All entries in the BTB are invalidated, causing all initial branches to be predicted by the static, rather than dynamic, branch prediction units. For additional information, refer to the sections on branch prediction in the chapter entitled "The Fetch, Decode, Execute Engine" on page 61.
Prefetch queue	The prefetch queue is cleared, so there are no instructions available to the instruction pipeline.
Instruction decode queue	The instruction decode queue is invalidated, so there are no micro-ops available to be placed in the instruction pool for execution.
Instruction pool	The instruction pool is cleared, so there are no micro-ops available for execution.
CR0	Contains 60000010h. The processor is in real mode with paging disabled.
CR4	Software Features register. Contains 00000000h. All post-486 (Pentium and Pentium Pro) software features are disabled.

Chapter 4: Processor Startup

Table 4-1: Effects of Reset on CPU (Continued)

Effect	Result
DTLB and ITLB	Data and Instruction Translation Lookaside Buffers. All DTLB and ITLB entries are invalidated. This has no initial effect because paging is disabled.
APIC	Advanced Programmable Interrupt Controller. Has been assigned an ID (see the section "Selection of Processor's Agent and APIC IDs" on page 51) and begins (after the BIST, if the BIST has been enabled) to negotiate with the APICs in the other processors to select the bootstrap processor (see "Selection of Bootstrap Processor (BSP)" on page 55). Recognition of all external interrupts is disabled.
EFLAGS register	Contains 00000002h. Recognition of external interrupts is disabled.
EIP	Extended Instruction Pointer register. Contains 0000FFF0h.
CS	Code Segment register. The visible part of the CS register contains F000h. The invisible part contains the following: base = FFFF0000h; limit = FFFFh; AR = present, R/W, accessed.
CR2	Page Fault Address register. Contains 00000000h. No effect.
CR3	Page Directory Base Address register. Contains 00000000h. No effect (because paging is disabled).
SS, DS, ES, FS, GS	Stack and data segment registers. Contain 00000000h.
EDX	Contains 000006xxh, where xx can be any 2 hex digits (see "EDX Contains Processor Identification Info" on page 55 for additional information).

Pentium Pro Processor System Architecture

Table 4-1: Effects of Reset on CPU (Continued)

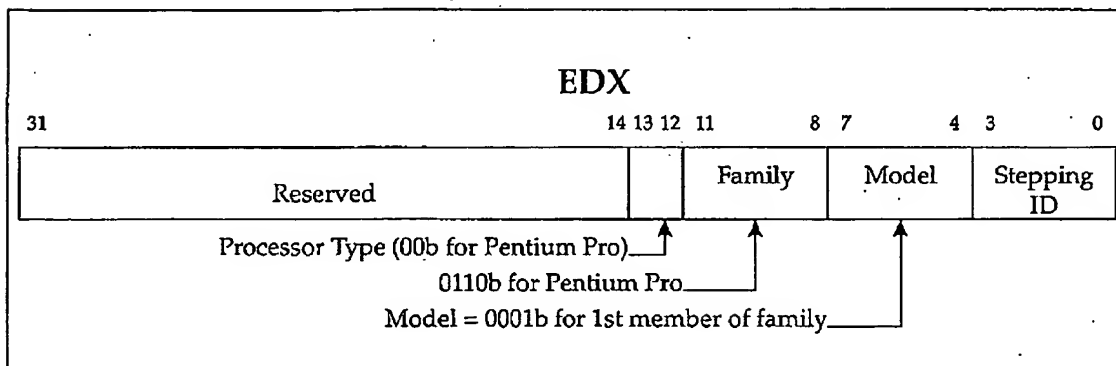
Effect	Result
EAX	Contains 00000000h if BIST not performed, and BIST result if BIST performed. See "Run BIST Option" on page 39.
EBX, ECX, ESI, EDI, EBP, ESP	Contain 00000000h.
LDTR	Local Descriptor Table Register. Visible part contains 0000h. Invisible part contains: base = 00000000h; limit = FFFFh. No effect (because processor is in real mode).
GDTR and IDTR	Global and Interrupt Descriptor Table Registers. Base = 00000000h. Limit = FFFFh. Access rights = present and R/W. GDTR contents has no effect because processor is in real mode. IDTR contents places real mode interrupt table at memory location 00000000h.
DR0 through DR3	Debug Breakpoint Address Registers. Contain 00000000h.
DR6	Debug Status Register. Contains FFFF0FF0h.
DR7	Debug Control register. Contains 000000400h. All four breakpoints are disabled.
Time Stamp Counter (TSC)	Contains 0000000000000000h. After removal of reset, incremented once for each processor clock.
Performance Counter registers and Event Select registers	Cleared to zero. Counting of Performance Events is disabled.
Other Model-Specific Registers (MSRs)	Undefined.
Memory Type and Range Registers (MTRRs)	Disabled, causing all 4GB of memory space to be treated as UC (UnCacheable) memory.
Machine Check Architecture (MCA)	In undefined state.

Chapter 4: Processor Startup

EDX Contains Processor Identification Info

After reset, the EDX register contains 000006xxh, where xx can be any 2 hex digits. Figure 4-1 on page 55 illustrates the EDX contents.

Figure 4-1: EDX Contents after Reset



State of Caches and Processor's Ability to Cache

Reset's setting of the CR0[CD] and CR0[NW] bits disables the processor's ability to cache. Although this bit setting permits it to perform lookups in the caches, all lookups result in misses (because the caches were cleared by reset).

The instruction streaming buffer (i.e., the prefetch queue), instruction decode queue, and instruction pool are empty, resulting in total starvation for the processor's fetch, decode and execute engines. The processor must therefore immediately begin fetching instructions from external memory.

Selection of Bootstrap Processor (BSP)

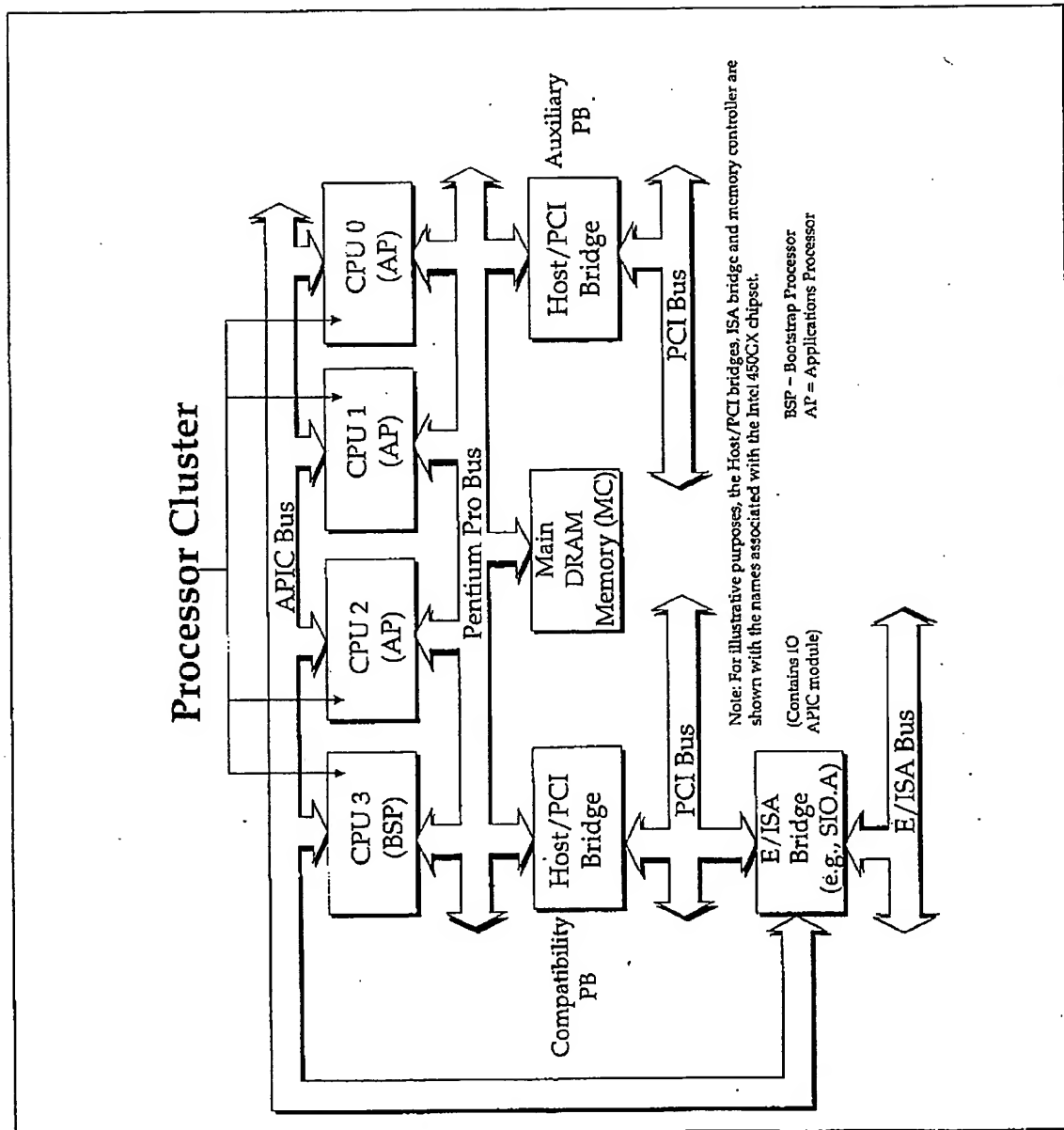
Introduction

Refer to Figure 4-2 on page 56. Upon the deassertion of reset (and the completion of a processor's BIST, if started), the processors within the cluster must negotiate amongst themselves to select the processor that will wake up and start fetching, decoding and executing the POST code from ROM memory. This processor is referred to as the Bootstrap Processor, or BSP. After the BSP is iden-

Pentium Pro Processor System Architecture

tified, the other processors, referred to as the applications processors, or APs, remain dormant (in the halt state) until they receive a startup message (via the APIC bus) from the BSP. The following section provides a detailed description of the BSP selection process. For a detailed discussion of the APIC, refer to the APIC section in MindShare's *Pentium Processor System Architecture* book (published by Addison-Wesley).

Figure 4-2: System Block Diagram



BSP Selection Process

APIC Arbitration Background

During output of any message on the APIC bus, the local APIC's 4-bit arbitration ID is inverted and driven onto APIC data bus line one (PICD1) serially, msb first. Multiple processors may start driving messages simultaneously. In this case, as each inverted bit of the arbitration ID is driven onto the bus, msb first, an electrical zero beats an electrical one. When a processor driving an electrical one sees an electrical zero on the line, it realizes that it has lost the arbitration and ceases to attempt transmission of its message. It waits until the current message transmission completes and then reattempts transmission of its message.

Startup APIC Arbitration ID Assignment

On the trailing-edge of reset, each processor's local APIC's arbitration ID is set to its APIC ID number (which is the same as the processor's agent ID). At start-up time, the net result is that the processor with an agent ID (and therefore an APIC arbitration priority ID) of 0h has the lowest APIC priority and the processor with highest numerical agent ID is the highest APIC priority. The final result is that the processor with the numerically highest agent ID will be the BSP (as described in the next section).

BSP Selection Process

1. After each processor's BIST completes (if it was started), the local APICs in all processors attempt to send their first BIPI (Bootstrap Inter-Processor Interrupt) message to all processors (including themselves) over the APIC bus.
2. The processor with the highest APIC arbitration priority (i.e., inverted agent ID) wins the arbitration and sends the first BIPI message to all of the processors (including itself). The processors that lose the arbitration must wait until the winner finishes issuing its BIPI message before they reattempt issuance of their BIPI messages.
3. The arbitration winner's BIPI message is received by all of the processors and the APIC ID (lower 4 bits of the 8-bit vector field in the message) of the winner is compared to the receiving processor's APIC ID. When the winner's APIC ID matches a processor's APIC ID, that processor sets the BSP bit in its APICBASE MSR (model-specific register). This identifies it as the bootstrap processor. All of the losers clear this bit in their respective APIC-

Pentium Pro Processor System Architecture

BASE MSRs, thereby identifying themselves as the applications processors, or APs.

4. The winner (i.e., the BSP) changes its rotating APIC priority level to 0 (the lowest APIC priority) and attempts to issue the FIPI (Final boot Inter-Processor Interrupt) message to all processors (including itself).
5. All of the processors that lost the first competition (the APs) attempt once again to transmit their BIPI messages and the winner of the previous arbitration (the BSP) attempts to transmit its FIPI message.
6. Because the BSP set its arbitration priority level to the lowest, it is guaranteed to lose the competition. One of the other processors will win.
7. As each of the APs is successful in acquiring APIC bus ownership and transmitting its BIPI message, it then adds one to its priority level, thereby making itself less important.
8. As each of the application processors (APs) in succession wins the bus and finishes broadcasting its BIPI, it then remains in the halt state until it subsequently receives a SIPI (Startup Inter-Processor Interrupt message) from the BSP at a later time.
9. After all of the APs have broadcast their BIPIs, the BSP will be successful in re-acquiring APIC bus ownership and will then broadcast its FIPI. Upon receiving its own FIPI, the BSP then begins fetching the POST code.

Once the BSP selection process has completed, the BSP initiates fetch, decode and execution of the ROM POST code starting at the power-on restart address selected at the trailing edge of reset (see "Power-On Restart Address Selection" on page 40).

Processor's Initial Memory Reads

Although the processor comes out of reset with caching disabled, the currently-available versions of the processor indulge in aberrant behavior when fetching the POST code from the boot ROM. Rather than performing a single-quadword read to get the first eight bytes of code from ROM starting at the power-on restart address, it proceeds as follows:

1. Assume that the power-on restart address selected at the trailing-edge of reset is 0FFFFFFF0h (4GB-16). This is the third quadword of the 32 byte memory line that starts at 0FFFFFFE0h.
2. The BSP initiates a 32 byte read from ROM starting at 0FFFFFFF0h (i.e., it acts as if its performing a cache line fill from memory starting at the critical quadword requested by the instruction prefetcher).
3. The four quadwords of the line from ROM are transferred back to the processor in the following order: 0FFFFFFF0h, 0FFFFFFF8h, 0FFFFFFFE0h, and

Chapter 4: Processor Startup

0FFFFFFF8h. The processor's bus interface keeps the first quadword and discards the remaining three.

4. The processor then initiates another 32 byte memory read to refetch the same line from memory, but this time it starts at the next quadword (in toggle mode order), 0FFFFFFF8h. The first quadword is kept and the others are discarded.

In other words, although the processor is not caching the code, it is performing line reads. Strange! When the startup firmware gets to the point where it enables caching by clearing the CR0[CD] and CR0[NW] bits, the processor stops doing this and starts behaving itself—it still doesn't cache (because the MTRRdefType register indicates that all of memory is uncacheable), but it no longer indulges in the wasteful 32 byte reads. 32 bytes reads will not be performed until the MTRRs are initialized defining some memory as cacheable.

How APs are Started

When the BSP has completed execution of the POST and has configured and enabled the devices necessary to read the OS from a mass storage device, it executes the OS boot to begin the process of reading the OS into memory and, after doing so, passes control to the OS startup code.

Once the OS startup code has been read into memory, the boot program jumps to and starts executing it. The OS startup code is responsible for reading the remainder of the OS into memory.

Uni-Processor OS

If the OS is not capable of recognizing the existence of the APs, they remain dormant and are never used (in other words, they suck power and are a waste of money).

SMP OS

If the OS is a Symmetrical Multi-Processing, or SMP, OS, it then must detect the presence of the other processors (the APs), place tasks in memory for them to execute, and pass the start address of these programs to each of them. The section that follows defines the AP detection process. Note that the Intel Multi-Processing spec calls for the BIOS, rather than the SMP OS to perform the AP

Pentium Pro Processor System Architecture

detection process and to supply the SMP OS with this information (typically, in CMOS memory).

AP Detection

The SMP OS kernel executing on the BSP can detect the presence of APs by performing the following sequence:

1. Write a program into main memory that, when executed, causes a program-defined memory location to be set to a specific value.
2. Make sure that the memory location is cleared.
3. Setup the BSP's local APIC timer to generate an interrupt in approximately 100ms.
4. Command the BSP's local APIC to issue a Startup IPI (SIPI) message to one of the other processor's in the cluster. The SIPI message includes the start address of the program previously written into main memory.
5. Poll the memory location to see if the targeted processor has changed the value (in other words, whether or not it has received the SIPI and initiated program execution). The BSP's program should spin on this step until either the location is changed by the AP or the timer interrupt occurs.
 - If the timer interrupt occurs before the memory location is changed, the assumption is made that the target AP processor isn't present to receive the SIPI.
 - If the location is changed within the timeout period, the target processor received the SIPI and initiated execution of the program deposited in memory by the BSP. Disable the APIC timer.
6. Repeat steps 2 through 5 for each of the possible APs that may be present.

AP Task Assignment

Assuming that one or more APs are detected using this procedure, the SMP OS kernel can now deposit an initialization program in memory and have each AP execute it to switch them into protected mode and initialize all of MTRRs so that they all view the 4GB of memory space the same way. Having done this, the SMP OS can begin assigning tasks (i.e., programs) to each of the APs using SIPIs to start their execution.

5

The Fetch, Decode, Execute Engine

The Previous Chapter

The previous chapter described the processor state at startup time. It also described the process that the processors engage in to select the processor that will fetch and execute the power-on self-test (POST), as well as how, after it has been loaded, an SMP OS can detect the additional processors and assign tasks for them to execute.

This Chapter

This chapter provides a detailed description of the processor logic responsible for instruction fetch, decode and execution.

The Next Chapter

In preparation for the discussion of the processor's caches, the next chapter introduces the register set that tells the processor its rules of conduct within various areas of memory. The Memory Type and Range Registers, or MTRRs, must be programmed after startup to define the various regions of memory within the 4GB memory space and how the processor core and caches must behave when performing accesses within each region. A detailed description of the MTRRs may be found in the appendix entitled "The MTRR Registers" on page 505.

Please Note

This chapter discusses the internal architecture of the Pentium Pro processor. It must be stressed that this information is based on the information released by Intel to the general development community. Since Intel is not in the business of giving away their intellectual property, it stands to reason that they haven't

Pentium Pro Processor System Architecture

revealed every detail of the processor's operation. The author has made every effort to faithfully describe the processor's operation within the bounds of the information released by Intel.

Introduction

Throughout this chapter, refer to the overall processor block diagram pictured in Figure 5-1 on page 63. Figure 24-7 on page 457 illustrates the block diagram with the MMX execution units added in.

At the heart of the processor are the execution units that execute instructions. The processor includes a fetch engine that attempts to properly predict the path of program execution and generates an on-going series of memory read operations to fetch the desired instructions.

If the processor didn't include an instruction cache, each of these memory read requests would have to be issued (on the processor's external bus) to external memory. The processor would suffer severe performance degradation for two reasons:

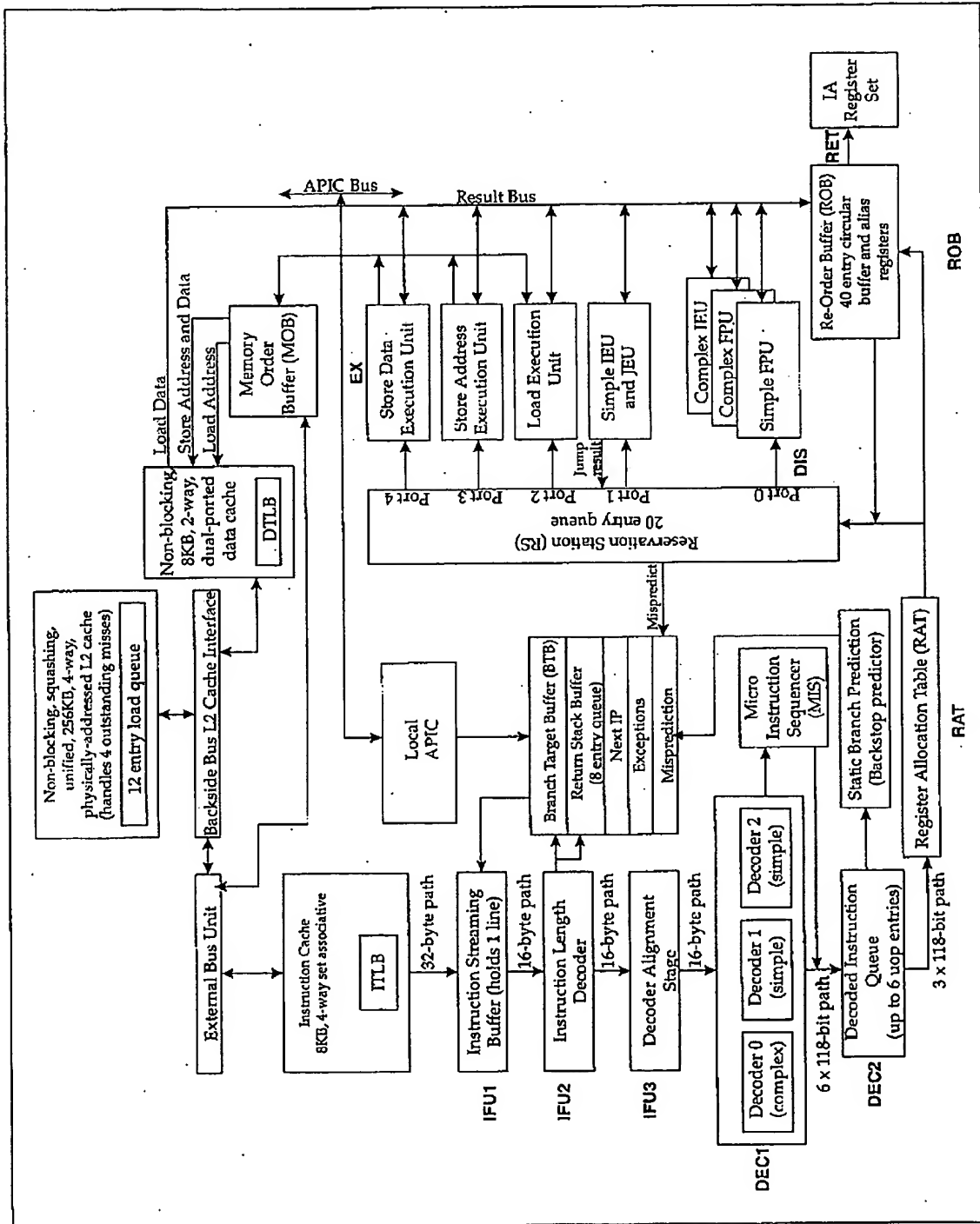
- The clock rate at which the processor generates external bus transactions is a fraction of the internal clock rate. This results in a relatively low-speed memory read to fetch the instructions.
- Secondly, the access time of main DRAM memory is typically even slower than the bus speed. In other words, the memory injects wait states into the bus transaction until it has the requested information available.

The high-speed (e.g., 150 or 200MHz) processor execution engine would then be bound by the speed of external memory accesses. It should be obvious that it is extremely advantageous to include a very high-speed cache memory on board the processor to keep copies of recently used (and, hopefully, frequently used) information (both code and data). Memory read requests generated by the processor core are first submitted to the cache for a lookup before being propagated to the external bus (in the event of a cache miss).

The Pentium Pro processor includes both a code and a data cache (the level one, or L1, caches). In addition, it includes an L2 cache tightly coupled to the processor core via a private bus. The processor's caches are disabled at powerup time, however. In order to realize the processor's full potential, the caches must be enabled. The following section describes the enabling of the caches.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-1: Overall Processor Block Diagram



Pentium Pro Processor System Architecture

Enabling the Caches

The processor's caches are disabled at startup (by reset) and are enabled when:

- CR0[CD] and CR0[NW] = 00b,
- and the targeted memory area is designated as cacheable by the MTRRs (memory type and range registers, described in the chapter entitled "Rules of Conduct" on page 119) and/or the selected page table entry (described in the chapter entitled "Paging Enhancements" on page 379). For more information on cache enabling, refer to the chapter entitled "The Processor Caches" on page 133.

Memory read requests are not issued to the code cache when:

- the caches are disabled or
- the target memory area is designated as non-cacheable by the MTRRs and/or the selected Page Table entry.

The following sections assume that the caches are enabled (via CR0) and the targeted memory area is designated cacheable (via the MTRRs and/or the selected page table entry). In this case, all code memory read requests generated by the instruction prefetcher are submitted to the code cache for a lookup. This discussion also assumes that the request results in a hit on the code cache. The code cache then supplies the requested information to the prefetcher.

Prefetcher

Issues Sequential Read Requests to Code Cache

In Figure 5-1 on page 63, the prefetcher interacts with the prefetch streaming buffer in the IFU1 pipeline stage. Although not directly pictured, it is represented by the Next IP block in the picture. The prefetcher continues issuing sequential access requests to the code cache unless one of the following events occurs:

- external interrupt.
- software exception condition.
- previously-fetched branch instruction results in a hit on the Branch Target Buffer (BTB) and the branch is predicted taken, or that static branch prediction results in a prediction of a branch to be taken.

Chapter 5: The Fetch, Decode, Execute Engine

In Intel's Pentium Pro documentation, the prefetch queue is referred to as the prefetch streaming buffer. The prefetch streaming buffer can hold 32 bytes of code (Please note that Intel's documentation states that it holds two 16-byte lines. In reality, it holds one line divided into two 16-byte blocks.). Assuming that caching is enabled, the prefetcher attempts to always keep the buffer full by issuing read requests to the code cache whenever the buffer becomes empty.

From the prefetch streaming buffer, the instructions enter the instruction pipeline. The series of stages that an instruction passes through from fetch to completion is referred to as the pipeline. The prefetch stage is the first stage of the pipeline.

Detailed Description of Prefetcher Operation

Figure 5-2 on page 66 illustrates a 64 byte area of memory (from memory location 0h through 3Fh) that contains a series of program instructions (for the time being, ignore the "S" and "C" designations). Notice that the length of the instructions vary. Some are one byte in length, some two bytes, etc. Assume that the processor's branch prediction logic predicts a branch to instruction one (the first instruction represented in grey). The prefetcher issues a request to the code cache for the line (consisting of memory locations 0-1Fh) that contains the first byte of the target instruction. These 32 bytes are loaded into the processor's prefetch streaming buffer (see Figure 5-3 on page 66).

In this example, the first 25-bytes (those that precede the branch target address) are unwanted instruction bytes and are discarded by the prefetcher. It should be noted that, although the figures illustrate the boundary between each instruction, in reality the boundaries are not marked in the prefetch streaming buffer. At this point, this is just a "raw" code stream. The boundaries will be determined and marked at a later stage of the pipeline (see "Decode Stages" on page 74).

These "raw" instructions are fed to instruction pipeline 16-bytes at a time. Until another branch is predicted, the prefetcher requests the next sequential line from the code cache each time that the streaming buffer is emptied by the processor core.

Pentium Pro Processor System Architecture

Figure 5-2: Example Instructions in Memory

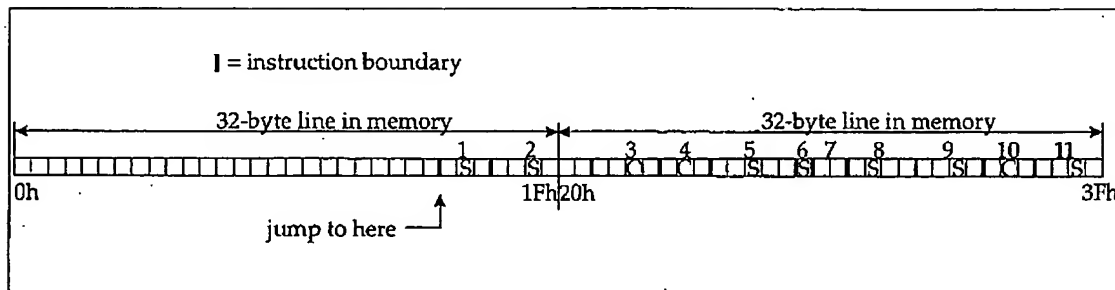
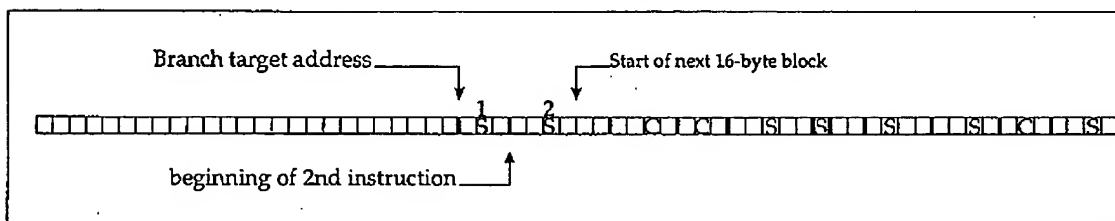


Figure 5-3: Contents of Prefetch Streaming Buffer Immediately after Line Fetched



In the example pictured in Figure 5-3 on page 66, the first 16-byte block is discarded, as are the first nine bytes of the second 16-byte block. Only instruction one is passed to the complex decoder (see Figure 5-4 on page 67) in the current clock. The complex decoder is referred to as decoder zero, while the two simple decoders are decoders one and two.

The second instruction cannot be passed to a decoder yet because it overflows into the second 16-byte block. In the next clock, the second 16-byte block is accessed, supplying the remainder of instruction two. The prefetcher always attempts to pass three instructions to decoders zero, one and two in strict program order (in the same clock). The earliest instruction in the next group of three instructions is aligned with decoder 0, while next two in-line instructions are aligned with decoders 1 and 2, respectively. The decoders are pictured in Figure 5-4 on page 67. Instruction two is therefore lined up with decoder 0, while instructions three and four are lined up with decoders 1 and 2, respectively. However, instructions 3 and 4 are both complex and cannot be handled by the simple decoders, so only instruction 2 is passed to decoder 0 in this clock.

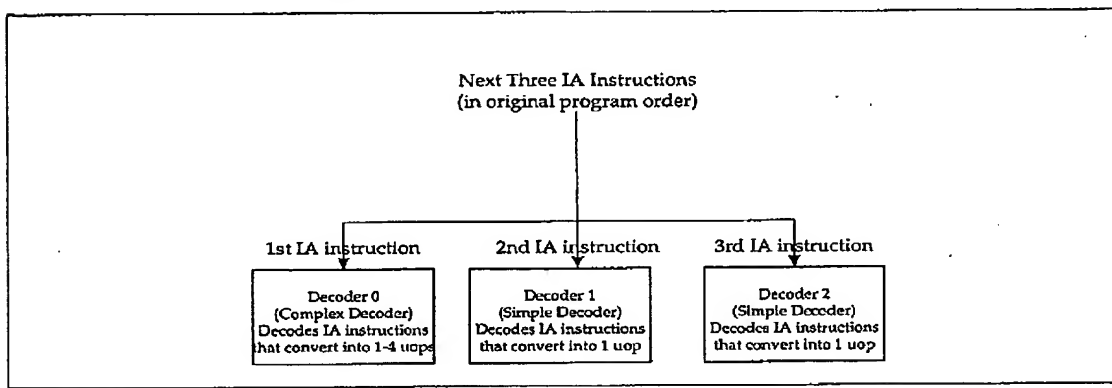
The next three instructions that have not yet been decoded (instructions 3, 4, and 5) are aligned with decoders 0 through 2, respectively, and consist of a complex/complex/simple series. While instruction 3 can be handled by decoder 0, instruction 4 cannot be handled by decoder 1. Since instructions are always

Chapter 5: The Fetch, Decode, Execute Engine

decoded in strict program order, instruction 5 cannot be decoded before instruction 4, so only instruction 3 is passed to decoder 0 in this clock.

In the next clock, the next three instructions, (4, 5, and 6) are lined up with decoders 0 through 2, respectively. In other words, a complex/simple/simple instruction series is lined up with decoders 0 through 2, respectively. In this case, all three instructions can be decoded simultaneously during the current clock.

Figure 5-4: The Three Instruction Decoders



Brief Description of Pentium Pro Processor

The processor:

1. Fetches IA instructions from memory in strict program order.
2. Decodes, or translates, them (in strict program order) into one or more fixed-length RISC instructions known as micro-ops, or uops.
3. Places the micro-ops into an instruction pool in strict program order.
4. Until this point, the instructions have been kept in original program order. This part of the pipeline is known as the in-order front end. The processor then executes the micro-ops in any order possible as the data and execution units required for each micro-op become available. This is known as the out-of-order (OOO) portion of the pipeline.
5. Finally, the processor commits the results of each micro-op execution to the processor's register set in the order of the original program flow. This is the in-order read-end.

Pentium Pro Processor System Architecture

Beginning, Middle and End

In-Order Front End

In the first seven stages of the instruction pipeline (the prefetch, decode, RAT and ROB stages), the instructions are kept in strict program order. The instructions are fetched, decoded into micro-ops, stored in the instruction decode queue and are then moved into the ROB in strict program order.

Out-of-Order (OOO) Middle

Once the micro-ops are placed in the instruction pool (i.e., the ROB) in strict program order, they are executed out-of-order, one or more at a time (up to five instructions can be dispatched and start execution simultaneously), as data and execution units for each micro-op become available. As each micro-op in the instruction pool completes execution, it is marked as ready for retirement and its results are retained in the micro-op's entry in the instruction pool. Intel has implemented features that permit the processor to execute instructions out-of-order:

- **Register aliasing.** Eliminates false dependencies created by the small IA register set.
- **Non-blocking data and L2 caches.** Permits the processor to continue to make progress when cache misses occur.
- **Feed forwarding.** Result of one micro-op's execution are immediately made available to other micro-ops that require the results in order to proceed with their own execution.

In-Order Rear End

The retirement unit (in the RET1 and RET2 stages) is constantly testing the three oldest micro-ops in the instruction pool to determine when they have completed execution and can be retired in original program order. As each group of three micro-ops is retired, the results of their execution is "committed" to the processor's IA register set. The micro-ops are then deleted from the pool and the pool entries become available for assignment to other micro-ops.

Intro to the Instruction Pipeline

Figure 5-5 on page 69 illustrates the main instruction pipeline. Table 5-1 on page 69 provides a basic description of each stage, while the sections that follow the table provide a detailed description of each.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-5: Instruction Pipeline

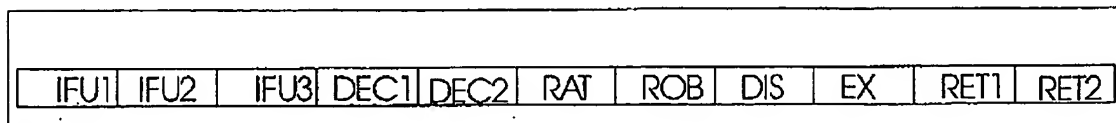


Table 5-1: Pipeline Stages

Stage	Description
IFU1	Instruction Fetch Unit stage 1. Load a 32-byte line from the code cache into the prefetch streaming buffer.
IFU2	Instruction Fetch Unit stage 2. Mark instruction boundaries. In the IFU2 stage, two operations are performed simultaneously: <ul style="list-style-type: none"> The boundaries between instructions within a 16-byte block are identified. Intel documents frequently refer to this as a line of information, but the term line is typically applied to the size of a block of information in a cache. To avoid possible confusion, the author therefore has decided not to refer to this as a line. If any of the instructions within the 16-byte block are branches, the memory addresses that they were fetched from are presented to the BTB (Branch Target Buffer) for branch prediction. Refer to "Description of Branch Prediction" on page 108.
IFU3	Instruction Fetch Unit stage 3. Align the instructions for presentation to the appropriate decoders (see Figure 5-4 on page 67) in the next stage.
DEC1	Decode stage 1. Translate (i.e., decode) the instructions into the corresponding micro-op(s). Up to three instructions can be decoded simultaneously (if they are aligned with a decoder that can handle the instruction; see Figure 5-4 on page 67).
DEC2	Decode Stage 2. Pass the micro-ops to the decoded instruction queue. Since some IA instructions (e.g., string operations) translate into rather large streams of micro-ops and the decoded instruction queue can only accept six micro-ops per clock, it should be noted that this stage may have to be repeated a number of times (i.e, it may last for a number of processor clock ticks).

Pentium Pro Processor System Architecture

Table 5-1: Pipeline Stages (Continued)

Stage	Description
RAT	<p>Register Alias Table and Allocator stage. The processor determines if a micro-op references any source operands. If it does, the processor determines if the source operand should be taken from a real IA register or from an entry in the ROB. The latter case will be true if a micro-op previously placed in the ROB (from earlier in the code stream) will, when executed, produce the result required as a source operand by this micro-op. In this case, the source field in this micro-op is adjusted to point to the respective ROB entry.</p> <p>As an example, a micro-op earlier in the code stream may, when executed, place a value in the EAX register. In reality, when the processor executes this micro-op, it places the value, not in the EAX register, but rather in the ROB entry occupied by the micro-op. A micro-op later in the code stream that requires the contents of EAX as a source would be adjusted in the RAT stage to point to the result field of the ROB entry occupied by the earlier micro-op, rather than to the real EAX register.</p>
ROB	<p>ReOrder Buffer (ROB) stage. Move micro-ops from the RAT/Allocator stage to the next sequential three ROB entries at the rate of three per clock. If all of the data required for execution of a micro-op is available and an entry is available in the RS (Reservation Station) queue, also pass a copy of the micro-op to the RS. The micro-op then awaits the availability of the appropriate execution unit.</p>
DIS	<p>Dispatch stage. If not already done in the previous stage (because all of the data required by the micro-op was not available or an entry wasn't available in the RS), copy the micro-op from the ROB to the RS, and then dispatch it to the appropriate execution unit for execution.</p>
EX	<p>Execution stage. Execute micro-op. The number of clocks that an instruction takes to complete execution is instruction dependent. Most micro-ops can be executed in one clock.</p>
RET1	<p>Retirement stage 1. When a micro-op in the ROB has completed execution, all conditional branches earlier in the code stream have been resolved, and it is determined that the micro-op should have been executed, it is marked as ready for retirement.</p>

Chapter 5: The Fetch, Decode, Execute Engine

Table 5-1: Pipeline Stages (Continued)

Stage	Description
RET2	Retirement stage 2. Retire instruction. When the previous IA instruction has been retired and all of the micro-ops associated with the next IA instruction have completed execution, the real IA register(s) affected by the IA instruction's execution are updated with the instruction's execution result (this is also referred to as "committing" the results to the machine state) and the micro-op is deleted (i.e., retired) from ROB. Up to three micro-ops are retired per clock in strict program order.

In-Order Front End

Instruction Fetch Stages

The first three stages of the instruction pipeline are the instruction fetch stages. The Intel documentation states that this takes 2.5 clock periods, but it doesn't define which clock edges within that 2.5 clock period define the boundaries between the three stages.

IFU1 Stage: 32-Byte Line Fetched from Code Cache

The instruction pipeline feeds from the bottom end of the prefetch streaming buffer, requesting 16 bytes at a time. When the buffer is emptied, the prefetch logic issues a request (during the IFU1 stage) to the code cache for the next sequential line. The prefetcher issues a 32-byte-aligned address to the code cache and a 32-byte block of "raw" code (the next sequential line) is loaded into the prefetch streaming buffer by the code cache.

IFU2 Stage: Marking Boundaries and Dynamic Branch Prediction

During the IFU2 stage, the boundaries between instructions within the first 16-byte block are identified and marked. In addition, if any of the instructions are branches, the memory addresses that they were fetched from (for up to four branches within the 16-byte block) are presented to the branch target buffer (BTB) for branch prediction. For a detailed discussion of dynamic branch prediction, refer to the section entitled "Description of Branch Prediction" on

Pentium Pro Processor System Architecture

page 108. Assuming that none of the instructions are branches or that none of the branches are predicted taken, code fetching will continue along the same sequential memory path.

IFU3 Stage: Align Instructions for Delivery to Decoders

Refer to Figure 5-6 on page 73. The processor implements three decoders that are used to translate the variable-length IA instructions into fixed-length RISC instructions referred to as micro-ops. Each micro-op is 118 bits in length and is referred to as a triadic micro-op because it includes three elements (in addition to the RISC instruction itself):

- two sources
- one destination

Intel does not document the format of the micro-ops.

The first of the three decoders (decoder 0) is classified as a complex decoder and can translate any IA instruction that translates into between one and four micro-ops. The second and third decoders (decoders 1 and 2) are classified as simple decoders and can only translate IA instructions that translate into single micro-ops (note that most of the IA instructions translate into one micro-op). In general:

- Simple IA instructions of the register-register form convert to a single micro-op.
- Load instructions (i.e., memory data reads) convert to a single micro-op.
- Store instructions (i.e., memory data writes) convert to two micro-ops.
- Simple read/modify instructions convert into two micro-ops.
- Simple instructions of the register-memory form convert into two or three micro-ops.
- MMX instructions are simple.
- Simple read/modify/write instructions convert into four micro-ops.

Appendix D of Intel Application Note AP-526 (order Number 242816) contains the number of micro-ops that each IA instruction converts to.

In the IFU3 stage, the processor aligns the first of the next three sequential IA instructions (within the 16-byte block) with the complex decoder and, if it translates into no more than four micro-ops, delivers it to the decoder for translation into micro-ops. In addition, if the second and possibly the third of the three instructions are simple (i.e., translate into a single micro-op each), they can simultaneously be delivered to the simple decoders (decoders 1 and 2) for translation into micro-ops.

Chapter 5: The Fetch, Decode, Execute Engine

The complexity of each IA instruction and the order of the instructions dictates how many instructions can be delivered to the decoders (somewhere between one and three) for translation in a single clock. Consider the example scenarios described in Table 5-2 on page 73.

Figure 5-6: The Three Instruction Decoders

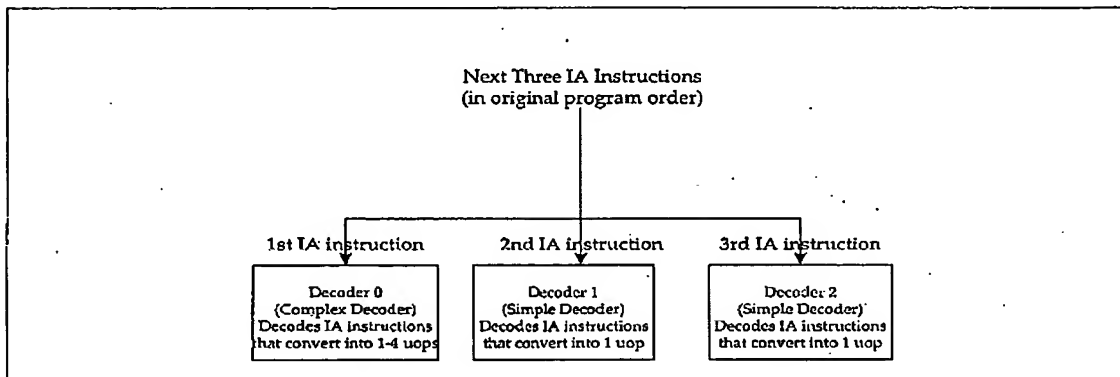


Table 5-2: Examples of IA Instruction Delivery to the Three Decoders

If next 3 Sequential IA Instructions in 16-byte Block are:	Instructions delivered to decoders in single clock
Simple, simple, simple	three
Simple, simple, complex	first two
Simple, complex, simple	first one
Simple, complex, complex	first one
Complex, simple, simple	three
Complex, simple, complex	first two
Complex, complex, simple	first one
Complex, complex, complex	first one

Pentium Pro Processor System Architecture

Decode Stages

After the IA instructions are aligned with the three decoders, they are submitted to one or more of the decoders (see “IFU3 Stage: Align Instructions for Delivery to Decoders” on page 72). There are two decode pipeline stages, DEC1 and DEC2, taking a total of 2.5 clocks to complete. The following sections discuss these two stages.

DEC1 Stage: Translate IA Instructions into Micro-Ops

In the DEC1 stage, between one and three IA instructions are submitted to decoders 0 through 2 for translation into micro-ops. The complex decoder can decode any IA instruction that is not greater than seven bytes in length and that translates into no more than four micro-ops. The simple decoders can decode any IA instruction that is not greater than seven bytes in length and that translates into a single micro-op. Most IA instructions are categorized as simple.

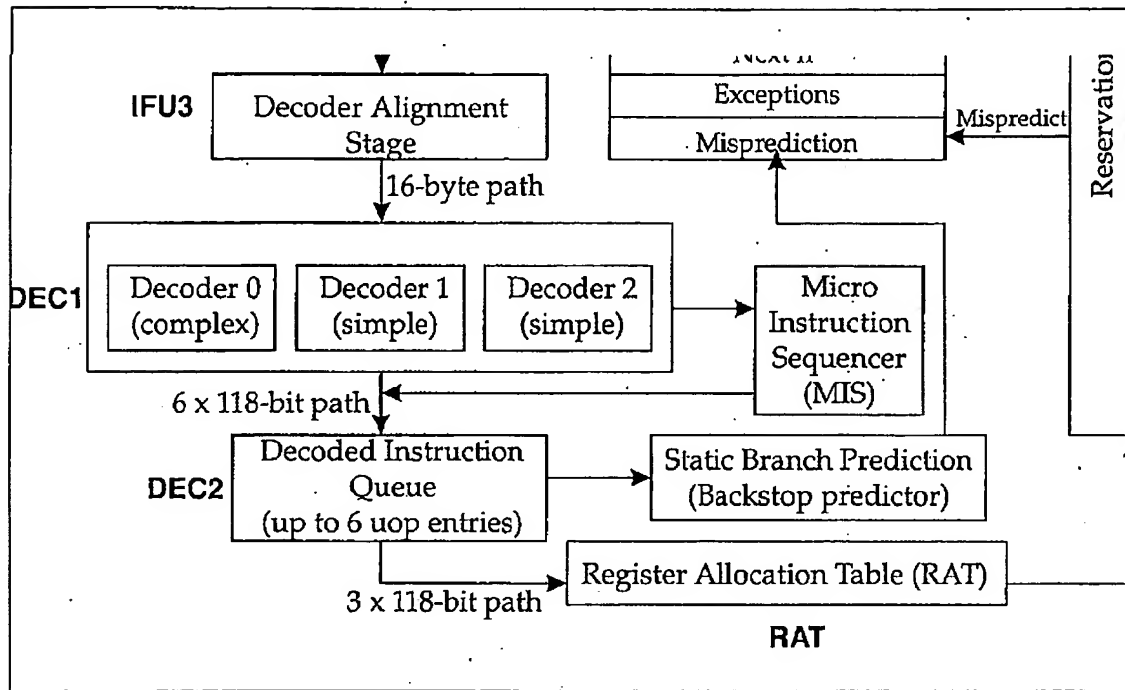
The best case throughput scenario is when a complex instruction is presented to decoder 0, and two simple instructions are simultaneously submitted to decoders 1 and 2. In this case, up to six micro-ops can be created in one clock (up to four from the complex instruction and one each for the two simple instructions). In the DEC2 stage (see “DEC2 Stage: Move Micro-Ops to ID Queue” on page 75), the micro-ops created are placed into the ID (instruction decode) queue in the same order as the IA instructions that they were translated from (i.e., in program order).

Micro Instruction Sequencer (MIS)

Refer to Figure 5-7 on page 75. Some IA instructions translate into more than four micro-ops and therefore cannot be handled by decoder 0. These instructions are submitted to the micro instruction sequencer (MIS) for translation. Essentially, the MIS is a microcode ROM that contains the series of micro-ops (five or more) associated with each very complex IA instruction. Some instructions (e.g., string operations) may translate into extremely large micro-op sequences. The micro-ops produced are placed into the ID queue in the DEC2 stage.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-7: Decoders and the Micro Instruction Sequencer (MIS)



DEC2 Stage: Move Micro-Ops to ID Queue

Two operations occur in the decode 2 stage:

1. The micro-ops produced by the decoders (or by the MIS) are placed in the ID queue in original program order. The best-case scenario is one where, in the DEC1 stage, the complex decoder is presented with an IA instruction that translates into four micro-ops and the two simple decoders are simultaneously presented with simple IA instructions that each create a single micro-op. In this case, six micro-ops are loaded into the ID queue simultaneously (note that the path between the DEC1 and DEC2 stages is 6 x 118-bits wide).
2. If any of the micro-ops in the ID queue represent branches, static branch prediction (for more information, refer to "Static Branch Prediction" on page 113) is applied to predict if the branch will be taken when executed or not.

Queue Micro-Ops for Placement in Pool. In the DEC2 stage, the micro-ops created by the three decoders or by the MIS are stored in the ID queue in original program order. The ID queue can hold up to six micro-ops and is pictured in Figure 5-7 on page 75.

Pentium Pro Processor System Architecture

Second Chance for Branch Prediction. As the micro-ops are deposited in the ID queue, the processor checks to see if any are branches. If any are, the processor's static branch prediction algorithm is applied to predict whether, when executed, the branch will be taken. For a detailed discussion of static branch prediction, refer to the section entitled "Static Branch Prediction" on page 113.

RAT Stage: Overcoming the Small IA Register Set

The IA register set only includes 16 registers that can be used to store data. They are the eight floating-point registers (FP0 through FP7) and the EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP registers. The small number of IA registers restricts the programmer to keeping a very small number of data variables close at hand in the registers. When these registers are all in use and the programmer needs to retrieve another data variable from memory, the programmer is forced to first copy the current contents of one of the IA registers back to memory before loading that register with new data from another area of memory. As an example, the following code fragment saves off the data in EAX before loading a new value into EAX:

```
mov [mem1], EAX
mov EAX, [mem2]
```

Due to the small register set, this sequence is very common in x86 programs. Ordinarily, the processor could not execute these two instructions simultaneously (because EAX must first be saved before it can be loaded with the new value). This is commonly referred to as a false register dependency that prevents the simultaneous execution of the two instructions.

However, the processor is designed to use another, hidden register as the register to be written by the second instruction and can therefore execute both instructions simultaneously. In other words, at the same time that the first instruction is reading the value from EAX to write into memory, the second instruction can read from memory and write into a hidden register rather than the real EAX register. The value in the hidden EAX can be copied into the actual EAX at a later time.

As another example, consider the following code fragment:

```
mov eax, 17
add mem, eax
mov eax, 3
add eax, ebx
```

Chapter 5: The Fetch, Decode, Execute Engine

Ordinarily, these four instructions would have to be executed in serial fashion, one after the other. The Pentium Pro can execute the first and the third instructions simultaneously, moving the values into two, separate hidden registers rather than into the `eax` register. Likewise, the second and the fourth instructions can then be executed simultaneously. The `eax` reference in the second instruction is changed to point to the hidden register that was loaded with the value 17 by the first instruction. The `eax` reference in the fourth instruction is changed to point to the hidden register that was loaded with the value 3 by the third instruction.

The processor includes a set of 40 hidden registers that can be used in lieu of any of the eight general purpose registers or any of the eight floating-point registers. This eliminates many false dependencies between instructions, thereby permitting simultaneous execution and resulting in better performance. During the register alias table (RAT) stage, the processor selects which of the 40 surrogate registers will actually be used by the micro-op when it is executed. The surrogate registers reside in the ROB (discussed in the next section).

ReOrder Buffer (ROB) Stage

After the micro-ops are produced by the decoders and have had their source fields adjusted in the RAT stage, they are placed in the 40-deep ROB in strict program order, at the rate of up to three per clock. Once placed in this instruction pool, the RS (reservation station) can copy multiple micro-ops from the pool (up to five simultaneously) and queue them up for delivery to the appropriate execution units. When micro-ops in the ROB have completed execution and are retired (described later), they are deleted from the ROB. These ROB entries are then available to move new micro-ops to the ROB from the ID queue. The sections that follow provide a detailed description of the ROB.

Instruction Pool (ROB) is a Circular Buffer

Refer to Figure 5-8 on page 79. During each clock, between one and three micro-ops (depending on how many micro-ops are currently in the ID queue) are moved from the top three locations (zero through two) of the ID queue to the Register Alias Table (RAT) stage. As stated earlier (see "RAT Stage: Overcoming the Small IA Register Set" on page 76), the processor adjusts the micro-ops' source fields to point to the correct registers (i.e., one of the real IA registers, or one of the 40 hidden registers in the ROB).

Pentium Pro Processor System Architecture

The ROB is implemented as circular buffer with 40 entries. A micro-op and, after it has completed execution, the result of its execution, are stored in each ROB entry. Initially, the ROB is empty and the start-of-buffer pointer and the end-of-buffer pointer both point to entry 0. As IA instructions are decoded into micro-ops, the micro-ops (up to three per clock) are placed in the ROB starting at entry 0 in strict program order. As additional micro-ops are produced by the decoders, the end-of-buffer pointer is incremented once for each micro-op. At a given instant in time, the end-of-buffer pointer points to where the next micro-op decoded will be stored, while the start-of-buffer pointer points to the oldest micro-op in the buffer (corresponding to the earliest instruction in the program).

Later, in the retirement stage, the retirement logic will retire (remove) the three oldest micro-ops from the pool and will increment the start-of-buffer pointer by three. The three pool entries that were occupied by the three micro-ops just retired are then available for new micro-ops.

To summarize, the logic is always adding new micro-ops to the end-of-buffer, incrementing the end-of-buffer pointer, and is also removing the oldest micro-ops from the start-of-buffer and incrementing the start-of-buffer pointer.

Pentium Pro Processor System Architecture

Out-of-Order (OOO) Middle

After the micro-ops are placed in the instruction pool (i.e., the ROB), the RS can copy multiple micro-ops from the pool in any order and dispatch them to the appropriate execution units for execution. The criteria for selecting a micro-op for execution is that the appropriate execution unit and all necessary data items required by the micro-op are available—the micro-ops do not have to be executed in any particular order. Once executed, the results of the micro-op's execution are stored in the ROB entry occupied by the micro-op.

In-Order Rear End (RET1 and RET2 Stages)

As already described in the previous sections, the processor fetches and decodes IA instructions in program order. The micro-ops produced by the decoders are placed into the ID queue in program order and are moved into the ROB in original program order. Once placed in the ROB, however, the micro-ops are dispatched and executed in any order possible. The execution results are stored in the respective ROB entries (rather than in the processor's actual register set).

When all upstream branches have been resolved (i.e., executed) and it has been established that a downstream micro-op that has already completed execution should have been executed, the micro-op is marked as ready for retirement (in the RET1 stage).

The retirement logic constantly checks the status of the oldest three micro-ops in the ROB (at the start-of-buffer) to determine when all three of them have been marked as ready for retirement. The micro-ops are then retired (in the RET2 stage), three at a time, in original program order. As each is retired, the micro-op's execution result is copied into the processor's real register set from the ROB entry and the respective ROB entry is then deleted.

Chapter 5: The Fetch, Decode, Execute Engine

Three Scenarios

To explain the relationship of—

- the prefetch streaming buffer,
- the decoders,
- the ID queue,
- the RAT,
- the ROB,
- the reservation station (RS),
- and the execution units,

the following sections have been included. They describe three example scenarios:

- The first assumes that reset has just been removed from the processor.
- The second assumes that the processor's caches have just been enabled.
- The third assumes that the processor's caches have been enabled for some time and the code cache contains lines of code previously fetched from memory.

Scenario One: Reset Just Removed

Immediately after reset is removed, the following conditions exist:

- Code cache and L2 cache are empty and caching is disabled.
- Prefetch streaming buffer is empty.
- ID queue is empty.
- ROB is empty.
- Reservation station (RS) is empty.
- All execution units are idle.
- The CS:IP registers are pointing to memory location FFFFFFF0h.

Starvation!

The ROB and the RS are empty, so the processor's execution units have no instructions to execute.

Pentium Pro Processor System Architecture

First Instruction Fetch

Because caching is disabled (CR0[CD] and CR0[NW] are set to 11b), the prefetcher bypasses the L1 code cache and the L2 cache and submits its memory read request directly to the external bus interface (this occurs in the IFU1 stage). This read request has the following characteristics:

- Memory address is FFFFFFFF0h, identifying the quadword (group of eight locations) consisting of memory locations FFFFFFFF0h through FFFFFFFF7h.
- When caching is disabled or when fetching code from an area of memory designated as non-cacheable (by the MTRRs or the selected page table entry), the prefetcher always requests eight bytes of information. In other words, it takes advantage of the full width of the processor's external data bus (64-bits).

First Memory Read Bus Transaction

In response to this 8-byte memory read request, the external bus interface arbitrates for ownership of the external bus and then initiates an 8-byte memory read. *Please note that, even though caching is disabled, the current implementations of the Pentium Pro processor initiate a 32-byte rather than an 8-byte read. The critical quadword (the one that starts at 0FFFFFFF0h) is returned by the boot ROM, followed by the remaining three quadwords that comprise the line. The processor passes the first quadword into the prefetch streaming buffer and discards the remaining three quadwords. When the prefetcher issues the request for the next sequential 8-bytes, the processor initiates another 32-byte read for the same line, keeps the first quadword and discards the next three returned. Whether or not future versions of the Pentium Pro will also indulge in this inefficient behavior is unknown at this time.*

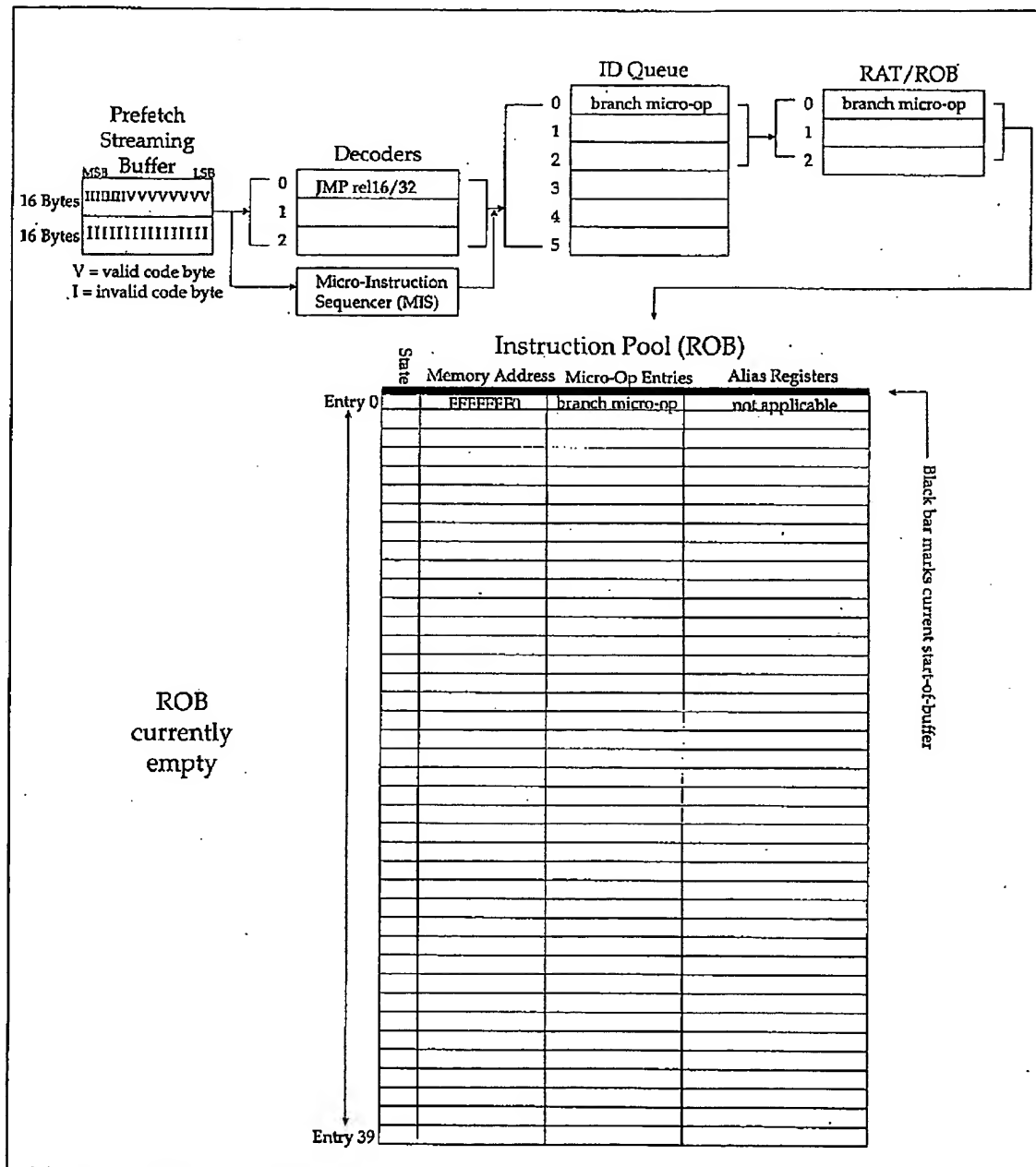
During this period of time, the processor core starvation continues (because the instructions have not percolated down through the instruction pipeline stages to be decoded and executed).

Eight Bytes Placed in Prefetch Streaming Buffer

Refer to Figure 5-9 on page 83. The addressed memory device presents the eight bytes back to the processor. The processor latches the data and places it in the first eight locations of the prefetch streaming buffer. The prefetcher immediately issues a request for the next sequential group of eight memory locations, FFFFFFFF8h through FFFFFFFFh and the bus interface initiates another external bus transaction.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-9: Scenario One Example—Reset Just Removed



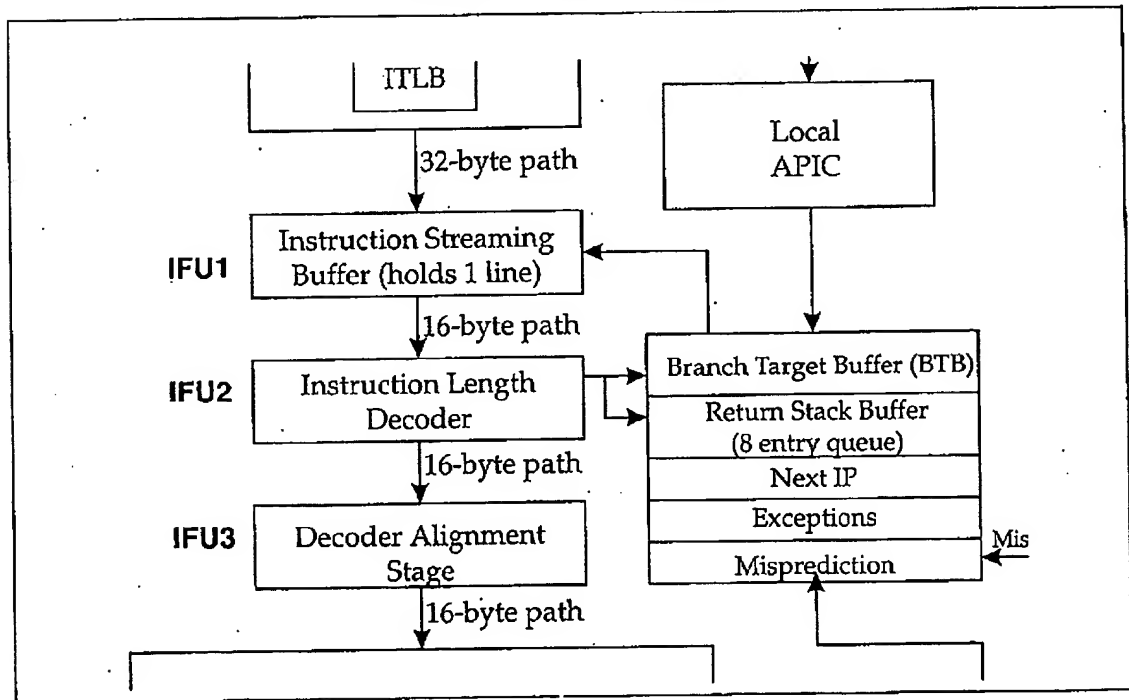
Pentium Pro Processor System Architecture

Instruction Boundaries Marked and BTB Checked

Refer to Figure 5-10 on page 84. In the IFU2 stage, the instruction boundaries within the eight-byte block are marked. In addition, if any of the instructions are branches (up to four of them), the memory address that it was fetched from is submitted to the BTB for a lookup. The BTB, a cache, was cleared by reset, so the lookup results in a BTB miss. In other words, the BTB has no history on the branch and therefore cannot predict whether or not it will be taken when it arrives at the jump execution unit (JEU) and is executed. Sequential fetching continues.

Realistically speaking, the first instruction fetched from the power-on restart address (FFFFFFFF0h) is almost always an unconditional branch to an address lower in memory (because FFFFFFFF0h is only 16 locations from the top of memory space).

Figure 5-10: The IFU2 Stage



Between One and Three Instructions Decoded into Micro-Ops

Refer to Figure 5-11 on page 86 and to Figure 5-9 on page 83. In the DEC1 stage, the first three IA instructions (assuming that there are three embedded within the first eight bytes) are submitted to the three decoders for translation into

Chapter 5: The Fetch, Decode, Execute Engine

micro-ops. The best case throughput scenario would be a simple/simple/simple or a complex/simple/simple sequence. This would produce between three and six micro-ops in one clock cycle.

In the DEC2 stage, the micro-ops are forwarded to the ID queue. In addition, if any of the micro-ops are branches, the static branch prediction logic decides whether or not to alter the fetch stream (i.e., to predict whether or not the branch will be taken when it arrives at the jump execution unit (JEU) and is executed).

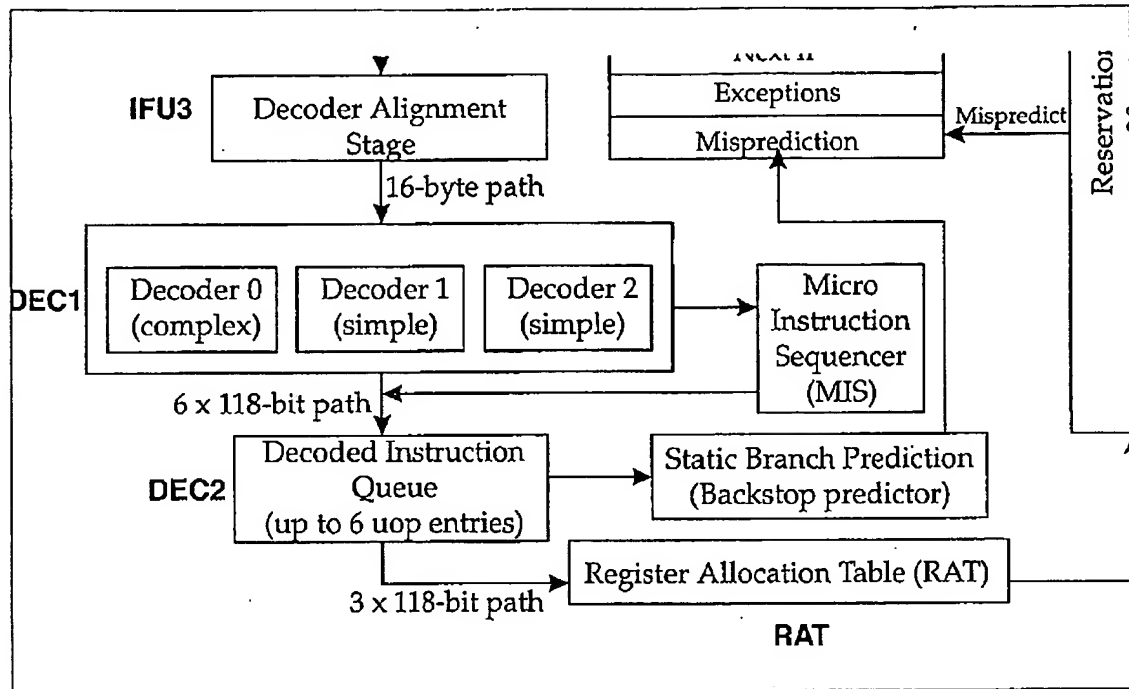
As mentioned earlier, the first instruction fetched after power-up is almost always an unconditional branch to an address lower in memory (the entry point of the system's power-on self-test, or POST). If this is the case, the static branch prediction logic predicts whether the branch will be taken when it is executed by the JEU (for more information, refer to "Static Branch Prediction" on page 113).

Assuming that the first instruction is an unconditional branch, it is predicted as taken (in the DEC2 stage). This has the following effects:

- The instructions in the IFU1 through DEC2 stages (i.e., in the prefetch streaming buffer and the ID queue) are flushed (the unconditional branch remains in the ID queue, however). The flushing of the instructions in the first five stages causes a "bubble" in the pipeline, resulting in a temporary decrease in performance.
- The branch target address is supplied to the prefetcher and is used to fetch the next instruction.
- Both the predicted branch target address and the fall-through address accompany the branch until it is executed by the JEU.

Pentium Pro Processor System Architecture

Figure 5-11: Decoders and the Micro Instruction Sequencer (MIS)



Source Operand Location Selected (RAT)

Refer to Figure 5-11 on page 86 and to Figure 5-9 on page 83. The branch instruction advances to the RAT stage where the processor determines if the micro-op references any source operands. If it does, the processor determines if the source operand should be supplied from a real IA register or from an entry in the ROB. The latter case will be true if a micro-op previously placed in the ROB (from earlier in the code stream) will, when executed, produce the result required as a source operand for this micro-op. In this case, the source field in this micro-op is adjusted to point to the respective ROB entry.

As an example, a micro-op earlier in the code stream may, when executed, place a value in the EAX register. In reality, when the processor executes this micro-op, it places the value, not in the EAX register, but rather in the ROB entry occupied by the micro-op. A micro-op later in the code stream that requires the contents of EAX as a source would be adjusted in the RAT stage to point to the result field of the ROB entry occupied by the earlier micro-op.

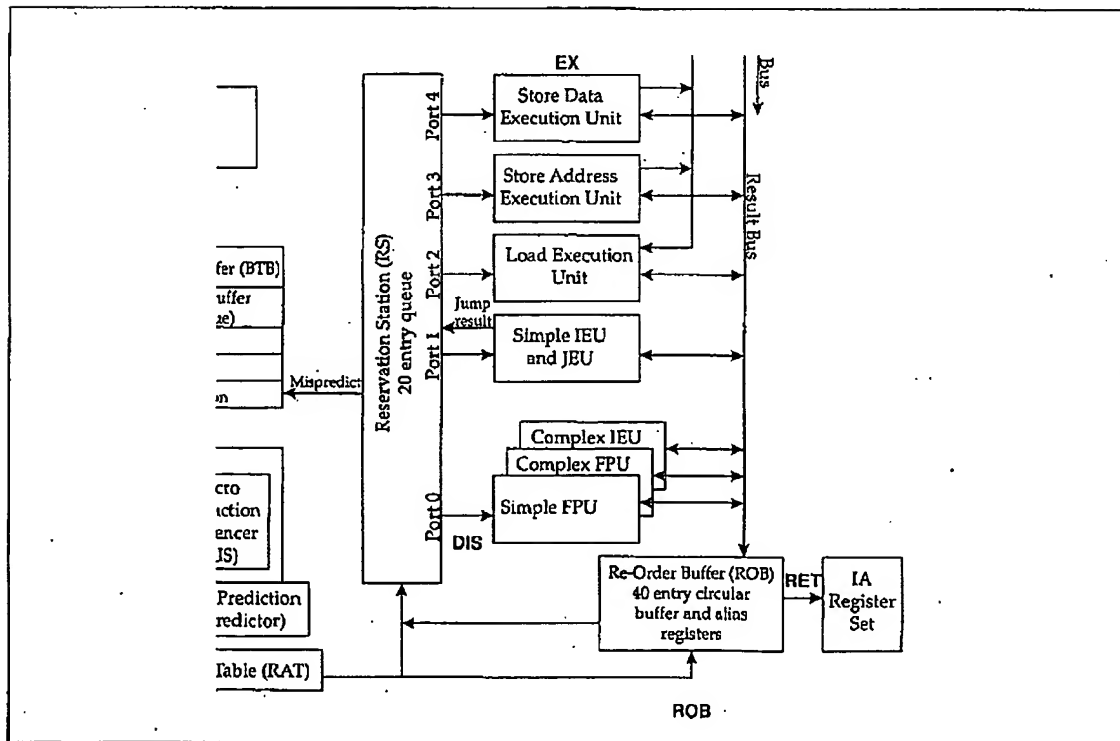
Chapter 5: The Fetch, Decode, Execute Engine

Micro-Ops Advanced to ROB and RS

Refer to Figure 5-12 on page 87. The reservation station (RS) is the gateway to the processor's execution units (see "The ROB, the RS and the Execution Units" on page 79). Basically, it is a queuing mechanism that, according to the Intel documentation, can queue up to 20 micro-ops to be forwarded to execution units as they become available. The documents don't define whether it is implemented as one 20-entry queue common to all five execution ports or as five separate queues (one for each execution port) each with four entries.

In the ROB stage, up to three micro-ops per clock are moved from the ID queue to the ROB in strict program order and are placed in the next three available ROB entries. In this example scenario, the branch micro-op in the ID queue is simultaneously moved to ROB entry zero and to the RS (because the RS has queue space available). In the event that there are no open RS queue entries, the micro-op is moved from the RAT to the ROB, but is not copied into the RS from the ROB until room becomes available (as the respective execution unit completes the execution of another micro-op).

Figure 5-12: The ReOrder Buffer (ROB) and The Reservation Station (RS)



Pentium Pro Processor System Architecture

Micro-Ops Dispatched for Execution

Refer to Figure 5-12 on page 87. The JEU is currently idle, so the branch micro-op is immediately dispatched from the RS to the JEU for execution during the EX (execution) stage and begins execution. In ROB entry zero, the branch micro-op is marked as executing (see Figure 5-9 on page 83).

Micro-Ops Executed

In this example, the micro-op is an unconditional branch, so the branch is taken when executed. The RS provides this feedback to the BTB where an entry is made indicating that the branch should be predicted taken the next time that is fetched. In addition, the branch target address is also stored in the BTB entry so that it will know where to branch to.

Since the branch was correctly predicted (by the static branch prediction logic during the DEC2 stage) and prefetching altered accordingly, the correct stream of instructions are behind the branch in the pipeline (albeit with a bubble in between). If the branch had been incorrectly predicted, the instructions that were prefetched after the branch, translated and placed in the ROB would be incorrect and must be flushed, along with those in pipeline stages earlier than the ROB (i.e., the RAT, ID queue, and prefetch streaming buffer).

Result to ROB Entry (and other micro-ops if necessary)

The branch micro-op has completed execution and its result is stored in the same ROB entry as the micro-op. The micro-op is marked completed in its respective ROB entry. Another micro-op currently awaiting execution in the ROB or RS may require the result produced by this micro-op. In this case, the result produced by the just-completed micro-op is now available to the stalled micro-op. That micro-op can then be scheduled for execution.

Micro-op Ready for Retirement?

It must be noted that micro-ops in the ROB are executed out-of-order (i.e., not in the original IA instruction order) as the required execution unit and any required data operands become available. A micro-op later in the program can therefore complete execution before instructions earlier in the program flow.

Consider the situation where there is a branch micro-op in the ROB from earlier in the program flow that has not yet been executed. When that branch is executed it may or may not alter the program flow. If it branches to a different area of memory, the micro-ops currently in the ROB that were prefetched from the

Chapter 5: The Fetch, Decode, Execute Engine

other path should not have been fetched, decoded and executed. Any micro-op already executed from the mispredicted path must therefore be deleted along with any results that it produced (in other words, as if it had never been executed).

A micro-op is not ready for retirement until it is certain that no branches from earlier in the program are currently in the ROB that, when executed, might result in a branch that would preclude the execution of this micro-op. When this condition is met, the branch micro-op in ROB entry zero is marked (in the RET1 stage) as ready for retirement.

Micro-Op Retired

In order to have the program execute in the order originally intended by the programmer, micro-ops must be retired in original program order. In other words, a micro-op that has completed execution and been marked as ready for retirement cannot be retired until all micro-ops that precede it in the program have been retired.

Retiring a micro-op means to permit its execution result to affect the processor's state (i.e., the contents of its real register set). Consider the following example:

```
add  edx, ebx
mov  eax, [0100]
cmp  eax, edx
jne  loop
```

The instructions that comprise this code fragment must alter and/or observe the processor's real register set in the following order:

1. `edx` must be changed to reflect the result of the addition.
2. `eax` must then be loaded with the four bytes from memory locations 00000100h through 00000103h in the data segment.
3. The values in `edx` and `eax` are then compared, altering the appropriate condition bits in the processor's EFLAGS register.
4. The conditional jump instruction then checks the EFLAGS[EQUAL] bit to determine whether or not to branch.

The process of retiring a micro-op involves:

- Permitting its result (currently stored only in the ROB entry) to be copied to the processor's real register set.
- Deleting the micro-op from the ROB
- The ROB entry then becomes available to accept another micro-op.

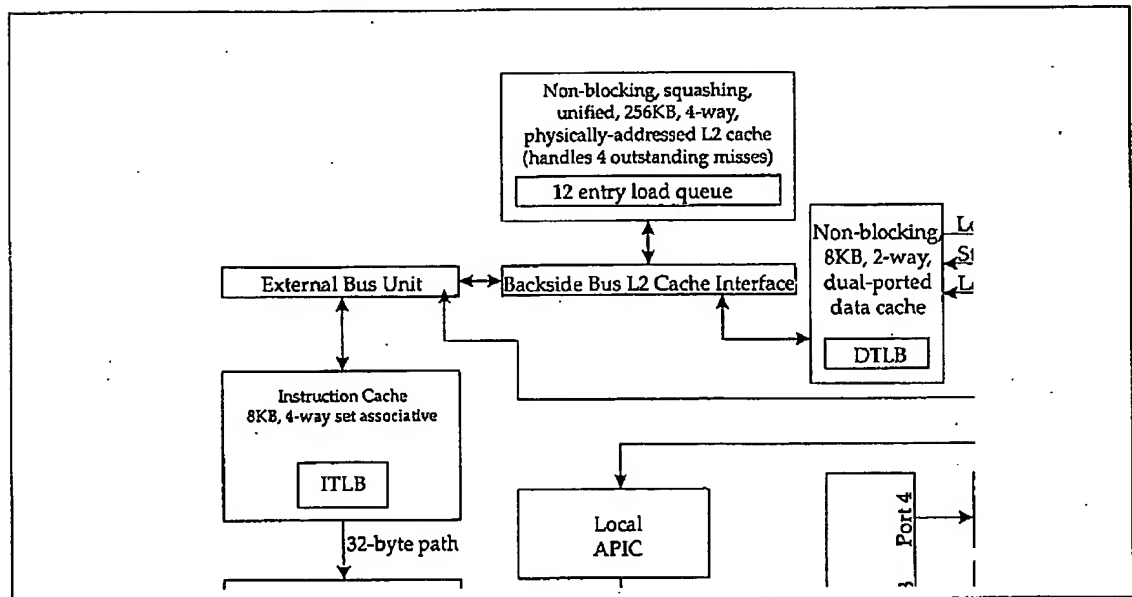
Pentium Pro Processor System Architecture

The processor is capable of retiring up to three micro-ops per clock during the RET2 stage.

Scenario Two: Processor's Caches Just Enabled

Assuming that the processor's caches have just been enabled (by clearing both CR0[CD] and CR0[NW] to zero), the caches are currently empty. Also assume that the processor's MTRR registers have been set up to define some memory as cacheable. When the prefetcher requests the next quadword, the quadword address is submitted to the code cache for a lookup. This results in a miss and the code cache issues a read request to the L2 cache (see Figure 5-13 on page 90) for the 32-byte code line that contains the requested quadword. This results in an L2 cache miss and the cache line request is issued to the processor's bus interface unit. The bus interface arbitrates for ownership of the external bus and initiates a 32-byte memory read request from main memory. The 32-byte line is returned to the processor in toggle-mode order (see "Toggle Mode Transfer Order" on page 171). The line is placed in the L2 cache and is also passed to the code cache where another copy is made. In addition, the line is immediately passed from the code cache to the prefetch streaming buffer. From this point, the processing of the code proceeds as already discussed starting in the section entitled "Instruction Boundaries Marked and BTB Checked" on page 84.

Figure 5-13: Code Cache and L2 Cache Miss



Chapter 5: The Fetch, Decode, Execute Engine

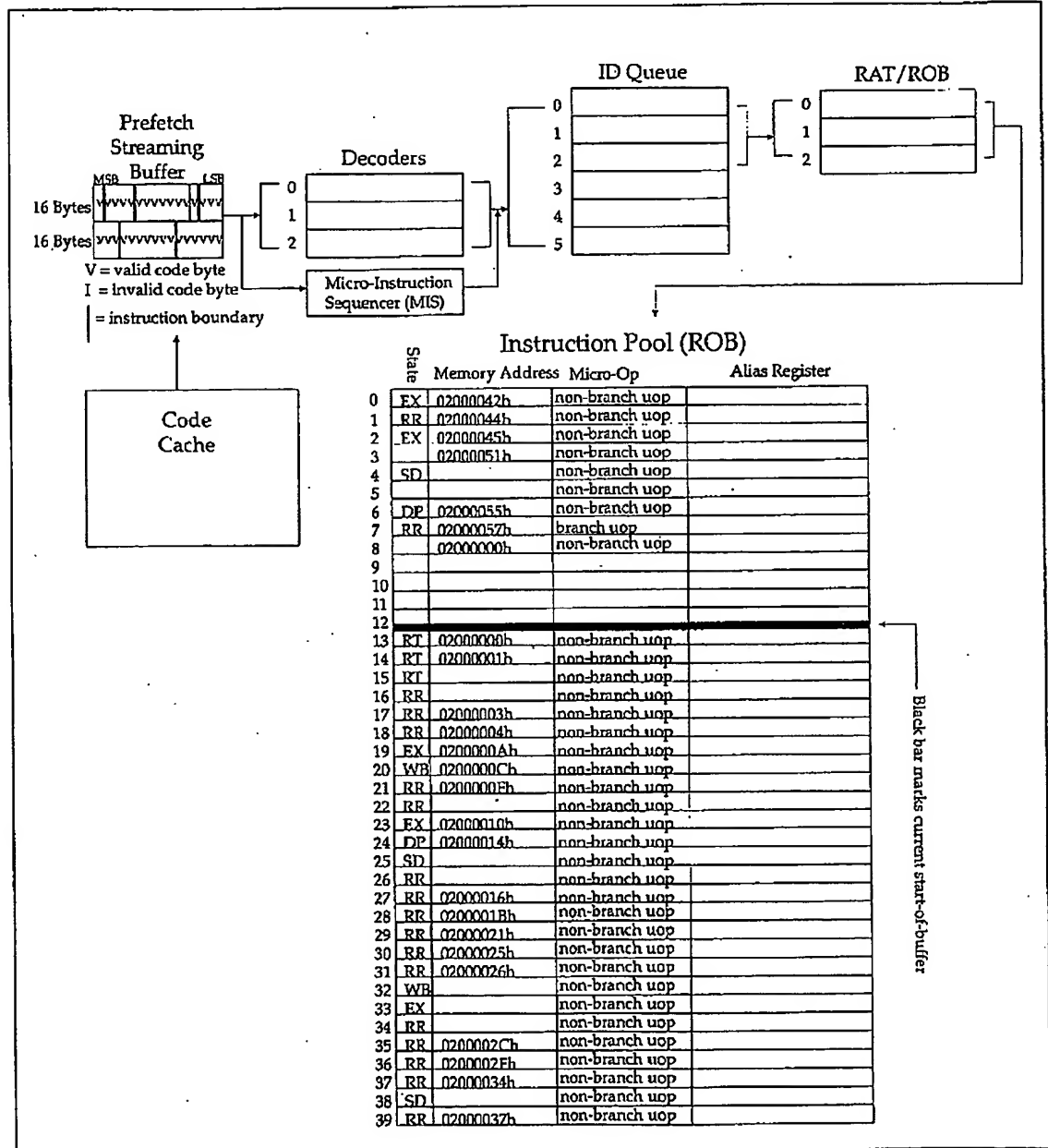
Scenario Three: Caches Enabled for Some Time

In this scenario, the processor's caches have been enabled for some time and the processor's instruction pipeline, including the ID queue, the ROB and the RS currently contain a stream of micro-ops in various stages of completion. Refer to Figure 5-14 on page 92 during this discussion.

Micro-ops (such as those illustrated in Figure 5-14 on page 92) are always placed in the ROB in original program order. The instruction in ROB entry 13 is the oldest instruction currently in the ROB, while the instruction in entry eight is the youngest. Instructions must be retired in strict program order, from oldest to youngest. The basic format of an entry (note that Intel does not provide this level of information, so this table is based on hopefully intelligent speculation) is introduced in Table 5-3 on page 93, while an entry-by-entry description is provided in Table 5-4 on page 94. Note that the micro-ops in Table 5-4 on page 94 are listed in original program order.

Pentium Pro Processor System Architecture

Figure 5-14: Scenario Three



Chapter 5: The Fetch, Decode, Execute Engine

Table 5-3: Basic Description of a ROB Entry

Element	Description
State	<p>A micro-op in the ROB can be in one of the following states:</p> <ul style="list-style-type: none">• SD: scheduled for execution. Indicates that the micro-op has been queued in the RS, but has not yet been dispatched to an execution unit.• DP: indicates that the micro-op is at the head of the RS dispatch queue for its respective execution port and is being dispatched to the respective execution unit for execution.• EX: the micro-op is currently being executed by its respective execution unit.• WB: the micro-op has completed execution and its results are being written back to the micro-op's ROB entry. In addition, if any other micro-ops are stalled waiting for the result, the result is forwarded to the stalled micro-op(s).• RR: the micro-op is ready for retirement.• RT: the micro-op is being retired.
Memory address	Indicates the start memory address of the IA instruction that generated the micro-op(s).
Micro-op	The micro-op is either a branch or a non-branch instruction.
Alias register	If a micro-op references one of the IA registers, the reference is re-directed to one of the 40 registers contained within the ROB (each ROB entry contains a register field that can store a value up to 80-bits wide (a floating-point value would be 80-bits wide, while integer values may be one, two, or four bytes wide).

Pentium Pro Processor System Architecture

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries

Entry	Explanation
13	This is the oldest micro-op in the ROB and, along with the next two micro-ops, is being retired in the current clock. The new start-of-buffer will then be 16 and entries 13, 14 and 15 will be available for new micro-ops (see Figure 5-15 on page 102). Entry 13 was the only micro-op decoded from the 1-byte IA instruction that was fetched from memory address 02000000h. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000001h (see entry 14).
14	This micro-op (and those in entries 15 and 16) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. Along with entries 13 and 15, it is being retired in the current clock. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
15	This micro-op (and those in entries 14 and 16) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. Along with entries 13 and 14, it is being retired in the current clock. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
16	This micro-op (and those in entries 14 and 15) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. It is ready for retirement and will be retired in the next clock (see Figure 5-16 on page 103) along with entries 17 and 18. The new start-of-buffer will then be 19 and entries 16, 17 and 18 will be available for new micro-ops. Entry 16 is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
17	This micro-op was the only one decoded from the 1-byte IA instruction that was fetched from memory location 02000003h. It is ready for retirement (see Figure 5-16 on page 103) and will be retired in the next clock along with entries 16 and 18. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000004h (see entry 18).
18	This micro-op was the only one decoded from the 6-byte IA instruction that was fetched from memory locations 02000004h through 02000009h. It is ready for retirement (see Figure 5-16 on page 103) and will be retired in the next clock along with entries 16 and 17. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Ah (see entry 19).

Chapter 5: The Fetch, Decode, Execute Engine

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
19	<p>This micro-op was the only one decoded from the 2-byte IA instruction that was fetched from memory locations 0200000Ah and 0200000Bh. It is currently being executed and, depending on the type of micro-op, may take one or more clocks to complete execution. When it completes execution, its state will change from EX to RR. It will be retired when:</p> <ul style="list-style-type: none"> • it has completed execution • its results have been written back to entry 19 • it and the micro-ops in entries 20 and 21 are ready for retirement • the micro-ops in entries 13 through 18 have been retired <p>It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Ch (see entry 20).</p>
20	<p>This micro-op was the only one decoded from the 3-byte IA instruction that was fetched from memory locations 0200000Ch through 0200000Eh. It has completed execution and its results are currently being written back to entry 20 (the micro-op is currently in the writeback stage). It will then be ready for retirement, but will not be retired until entries 19 and 21 are also ready for retirement (see entry 19). At that point, entries 19 through 21 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Fh (see entry 21).</p>
21	<p>This micro-op (and the one in entry 22) was decoded from the one-byte IA instruction that was fetched from memory location 0200000Fh. It is ready for retirement, but will not be retired until entries 19 and 20 are also ready for retirement (see entry 19). At that point, entries 19 through 21 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000010h (see entry 23).</p>
22	<p>This micro-op (and the one in entry 21) was decoded from the one-byte IA instruction that was fetched from memory location 0200000Fh. It will be retired when:</p> <ul style="list-style-type: none"> • it has completed execution • its results have been written back to entry 22 • it and the micro-ops in entries 23 and 24 are ready for retirement • the micro-ops in entries 13 through 21 have been retired <p>It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000010h (see entry 23).</p>
23	<p>This micro-op was the only one decoded from the 4-byte IA instruction that was fetched from memory locations 02000010h through 02000013h. It is still executing (and it may take more than one clock). After execution, it will enter the writeback stage where its execution results will be written back to entry 23. It will then be ready for retirement, but will not be retired until entries 22 and 24 are also ready for retirement (see entry 22). At that point, entries 22 through 24 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000014h (see entry 24).</p>

Pentium Pro Processor System Architecture

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
24	This micro-op and the ones in entries 25 and 26 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is currently being dispatched to its respective execution unit and will begin execution in the next clock. After execution, it will enter the writeback stage where its execution results will be written back to entry 24. It will then be ready for retirement, but will not be retired until entries 22 and 23 are also ready for retirement (see entry 22). At that point, entries 22 through 24 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).
25	<p>This micro-op and the ones in entries 24 and 26 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is currently in the RS and is scheduled for dispatch to its respective execution unit. After execution, it will enter the writeback stage where its execution results will be written back to entry 25. It will be retired when:</p> <ul style="list-style-type: none"> • it has completed execution • its results have been written back to entry 25 • it and the micro-ops in entries 26 and 27 are ready for retirement • the micro-ops in entries 13 through 24 have been retired <p>It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).</p>
26	This micro-op and the ones in entries 24 and 25 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is ready for retirement, but will not be retired until entries 25 and 27 are also ready for retirement (see entry 25). At that point, entries 25 through 27 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).
27	This micro-op was the only one decoded from the 5-byte IA instruction that was fetched from memory locations 02000016h through 0200001Ah. It is ready for retirement, but will not be retired until entries 25 and 26 are also ready for retirement (see entry 25). At that point, entries 25 through 27 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200001Bh (see entry 28).
28	This micro-op was the only one decoded from the 6-byte IA instruction that was fetched from memory locations 0200001Bh through 02000020h. It is ready for retirement, as are those in entries 29 and 30. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000021h (see entry 29).
29	This micro-op was the only one decoded from the 4-byte IA instruction that was fetched from memory locations 02000021h through 02000024h. It is ready for retirement, as are those in entries 28 and 30. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000025h (see entry 30).

Chapter 5: The Fetch, Decode, Execute Engine

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
30	<p>This micro-op was the only one decoded from the 1-byte IA instruction that was fetched from memory location 02000025h. It is ready for retirement, as are those in entries 28 and 29. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000026h (see entry 31).</p>
31	<p>This micro-op (and those in entries 32 through 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is ready for retirement and will be retired when:</p> <ul style="list-style-type: none"> the micro-op in entry 32 has written back its results to entry 32 and is ready for retirement the micro-op in entry 33 has completed execution, has written its results back to entry 33 and is ready for retirement the micro-ops in entries 13 through 30 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).</p>
32	<p>This micro-op (and those in entries 31, 33 and 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is writing its results back to entry 32 during the current clock and will be retired when:</p> <ul style="list-style-type: none"> it has written back its results to entry 32 and is ready for retirement the micro-op in entry 33 has completed execution, has written its results back to entry 33 and is ready for retirement the micro-ops in entries 13 through 30 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).</p>
33	<p>This micro-op (and those in entries 31, 32 and 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is executing and will be retired when:</p> <ul style="list-style-type: none"> It has completed execution and has written back its results to entry 33 and is ready for retirement the micro-op in entry 32 has written its results back to entry 32 and is ready for retirement the micro-ops in entries 13 through 30 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).</p>
34	<p>This micro-op (and those in entries 31, 32 and 33) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is ready for retirement, as are the micro-ops in entries 35 and 36. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).</p>

Pentium Pro Processor System Architecture

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
35	This micro-op was decoded from the 3-byte IA instruction that was fetched from memory locations 0200002Ch through 0200002Eh. It is ready for retirement, as are the micro-ops in entries 34 and 36. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Fh (see entry 36).
36	This micro-op was decoded from the 5-byte IA instruction that was fetched from memory locations 0200002Fh through 02000033h. It is ready for retirement, as are the micro-ops in entries 34 and 35. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000034h (see entry 37).
37	<p>This micro-op (and the one in entry 38) was decoded from the 3-byte IA instruction that was fetched from memory locations 02000034h through 02000036h. It is ready for retirement and will be retired (along with entries 38 and 39) when:</p> <ul style="list-style-type: none"> the micro-op in entry 38 has been dispatched, executed, its results have been written back to entry 37 and it's ready for retirement. the micro-ops in entries 13 through 36 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000037h (see entry 39).</p>
38	<p>This micro-op (and the one in entry 37) was decoded from the 3-byte IA instruction that was fetched from memory locations 02000034h through 02000036h. It is currently scheduled for dispatch to its respective execution unit. It will be retired (along with entries 37 and 39) when:</p> <ul style="list-style-type: none"> it has been dispatched, executed, its results have been written back to entry 37 and it's ready for retirement. the micro-ops in entries 13 through 36 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000037h (see entry 39).</p>
39	<p>This micro-op was decoded from the 11-byte IA instruction that was fetched from memory locations 02000037h through 02000041h. It is ready for retirement and will be retired (along with entries 37 and 38) when:</p> <ul style="list-style-type: none"> entry 38 has been dispatched, executed, its results have been written back to entry 38 and it's ready for retirement. the micro-ops in entries 13 through 36 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000042h (see entry 0).</p>

Chapter 5: The Fetch, Decode, Execute Engine

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
0	<p>This micro-op was decoded from the 2-byte IA instruction that was fetched from memory locations 02000042h through 02000043h. It is currently executing and will be retired (along with entries 1 and 2) when:</p> <ul style="list-style-type: none"> • it has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000044h (see entry 1).</p>
1	<p>This micro-op was decoded from the 1-byte IA instruction that was fetched from memory location 02000044h. It is ready for retirement and will be retired (along with entries 0 and 2) when:</p> <ul style="list-style-type: none"> • the micro-op in entry 0 has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000045h (see entry 2).</p>
2	<p>This micro-op was decoded from the 12-byte IA instruction that was fetched from memory locations 02000045h through 02000050h. It is currently executing and will be retired (along with entries 0 and 1) when:</p> <ul style="list-style-type: none"> • the micro-op in entry 0 has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000051h (see entry 3).</p>

Pentium Pro Processor System Architecture

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
3	<p>This micro-op (and the ones in entries 4 and 5) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is has not yet entered the RS and will be retired (along with entries 4 and 5) when:</p> <ul style="list-style-type: none"> the micro-op in entry 3 has been scheduled for dispatch (i.e., it has entered the RS), dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. the micro-ops in entries 13 through 2 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).</p>
4	<p>This micro-op (and the ones in entries 3 and 5) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is currently in the RS and has been scheduled for dispatch to its respective execution unit. It will be retired (along with entries 3 and 5) when:</p> <ul style="list-style-type: none"> the micro-op in entry 3 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. the micro-ops in entries 13 through 2 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).</p>
5	<p>This micro-op (and the ones in entries 3 and 4) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is has not yet entered the RS and will be retired (along with entries 3 and 4) when:</p> <ul style="list-style-type: none"> the micro-op in entry 3 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. the micro-ops in entries 13 through 2 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).</p>

Chapter 5: The Fetch, Decode, Execute Engine

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
6	<p>This micro-op was decoded from the 2-byte IA instruction that was fetched from memory locations 02000055h and 02000056h. It is being dispatched to its respective execution unit. It will be retired (along with entries 7 and 8) when:</p> <ul style="list-style-type: none"> the micro-op in entry 6 has been executed, its results have been written back to entry 6 and it's ready for retirement. the micro-op in entry 8 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 8 and it's ready for retirement. the micro-ops in entries 13 through 5 have been retired. <p>It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000057h (see entry 7).</p>
7	<p>This branch micro-op was decoded from the IA instruction that was fetched from memory starting at location 02000057h. It will be retired (along with entries 6 and 8) when:</p> <ul style="list-style-type: none"> the micro-op in entry 6 has been executed, its results have been written back to entry 6 and it's ready for retirement. the micro-op in entry 8 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 8 and it's ready for retirement. the micro-ops in entries 13 through 5 have been retired. <p>It is a branch micro-op, was predicted taken and therefore altered program flow. The next IA instruction was fetched from memory address 02000000h (see entry 8).</p>
8	<p>This was the only micro-op decoded from the 1-byte IA instruction that was fetched from memory address 02000000h. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000001h.</p>
9	empty
10	empty
11	empty
12	empty

Pentium Pro Processor System Architecture

Figure 5-15: Micro-Ops in Entries 13, 14 and 15 Have Been Retired

Instruction Pool (ROB)				
	State	Memory Address	Micro-Op	Alias Register
0	EX	02000042h	non-branch uop	
1	RR	02000044h	non-branch uop	
2	EX	02000045h	non-branch uop	
3		02000051h	non-branch uop	
4	SD		non-branch uop	
5			non-branch uop	
6	DP	02000055h	non-branch uop	
7	RR	02000057h	branch uop	
8		02000000h	non-branch uop	
9				
10				
11				
12				
13				
14				
15				
16	RR		non-branch uop	
17	RR	02000003h	non-branch uop	
18	RR	02000004h	non-branch uop	
19	EX	0200000Ah	non-branch uop	
20	WB	0200000Ch	non-branch uop	
21	RR	0200000Fh	non-branch uop	
22	RR		non-branch uop	
23	EX	02000010h	non-branch uop	
24	DP	02000014h	non-branch uop	
25	SD		non-branch uop	
26	RR		non-branch uop	
27	RR	02000016h	non-branch uop	
28	RR	0200001Bh	non-branch uop	
29	RR	02000021h	non-branch uop	
30	RR	02000025h	non-branch uop	
31	RR	02000026h	non-branch uop	
32	WB		non-branch uop	
33	EX		non-branch uop	
34	RR		non-branch uop	
35	RR	0200002Ch	non-branch uop	
36	RR	0200002Fh	non-branch uop	
37	RR	02000034h	non-branch uop	
38	SD		non-branch uop	
39	RR	02000037h	non-branch uop	

Black bar marks current start-of-buffer

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-16: Micro-Ops in Entries 16, 17 and 18 Have Been Retired

Instruction Pool (ROB)				
	State	Memory Address	Micro-Op	Alias Register
0	EX	02000042h	non-branch uop	
1	RR	02000044h	non-branch uop	
2	EX	02000045h	non-branch uop	
3		02000051h	non-branch uop	
4	SD		non-branch uop	
5			non-branch uop	
6	DP	02000055h	non-branch uop	
7	RR	02000057h	branch uop	
8		02000000h	non-branch uop	
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19	EX	0200000Ah	non-branch uop	
20	WB	0200000Ch	non-branch uop	
21	RR	0200000Fh	non-branch uop	
22	RR		non-branch uop	
23	EX	02000010h	non-branch uop	
24	DP	02000014h	non-branch uop	
25	SD		non-branch uop	
26	RR		non-branch uop	
27	RR	02000016h	non-branch uop	
28	RR	02000018h	non-branch uop	
29	RR	02000021h	non-branch uop	
30	RR	02000025h	non-branch uop	
31	RR	02000026h	non-branch uop	
32	WB		non-branch uop	
33	EX		non-branch uop	
34	RR		non-branch uop	
35	RR	0200002Ch	non-branch uop	
36	RR	0200002Fh	non-branch uop	
37	RR	02000034h	non-branch uop	
38	SD		non-branch uop	
39	RR	02000037h	non-branch uop	

Black bar marks current start-of-buffer

Pentium Pro Processor System Architecture

Memory Data Accesses—Loads and Stores

During the course of a program's execution, a number of memory data reads (i.e., loads) and memory data writes (i.e., stores) may be performed.

Handling Loads

Refer to Figure 5-17 on page 105. An IA load instruction decodes into a single load micro-op that specifies the address to be read from and the number of bytes to read from memory starting at that address. The load (i.e., a memory data read) micro-op is executed by the load execution unit (connected to port 2 on the RS). Unless prevented from doing so (by defining an area of memory such that speculative execution is not permitted; for more information see "Rules of Conduct" on page 119), loads can be executed speculatively in any order. In other words, a load micro-op from later in the program flow can be executed before one that occurs earlier in the program flow. A load instruction flows through the normal instruction pipeline until it is dispatched to the load execution unit. It has then entered the load pipeline stages. They are illustrated in Figure 5-18 on page 105 and detailed in Table 5-5 on page 106.

When executed, the load address is always compared to those of stores currently-posted in the posted-write buffer. If any of the stores in the posted-write buffer occur earlier in the program flow than the load and the store has updated the data to be read by the load, then the store buffer supplies the data for the read. This is referred to as store, or feed forwarding. For more information about loads, refer to "L1 Data Cache" on page 143.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-17: Load and Store Execution Units

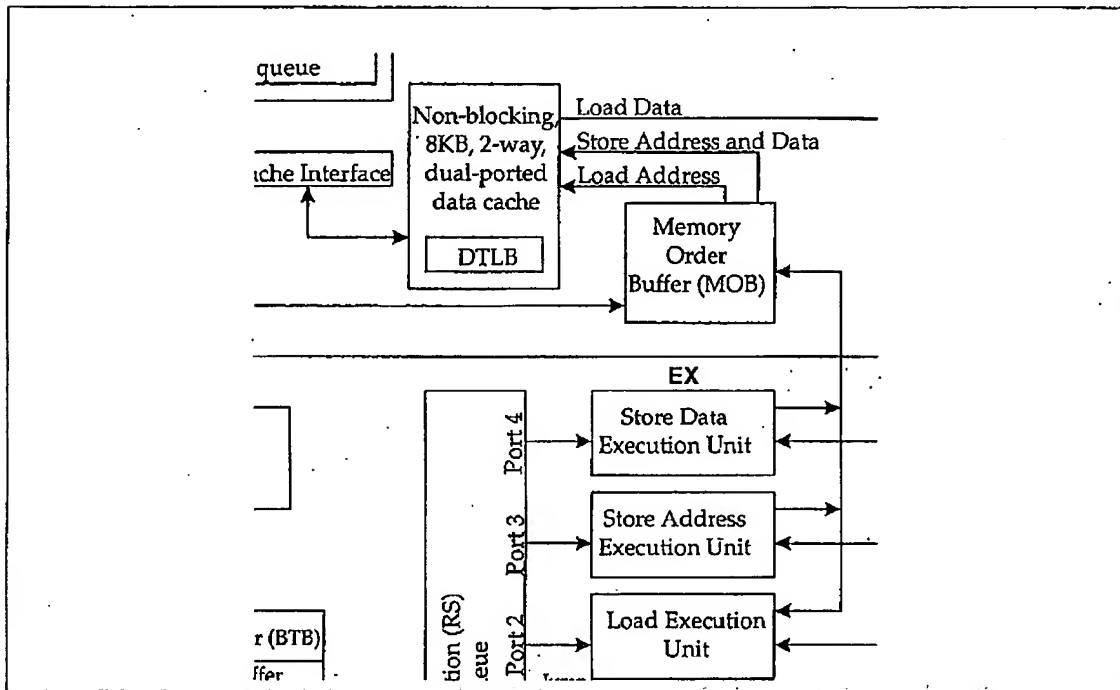
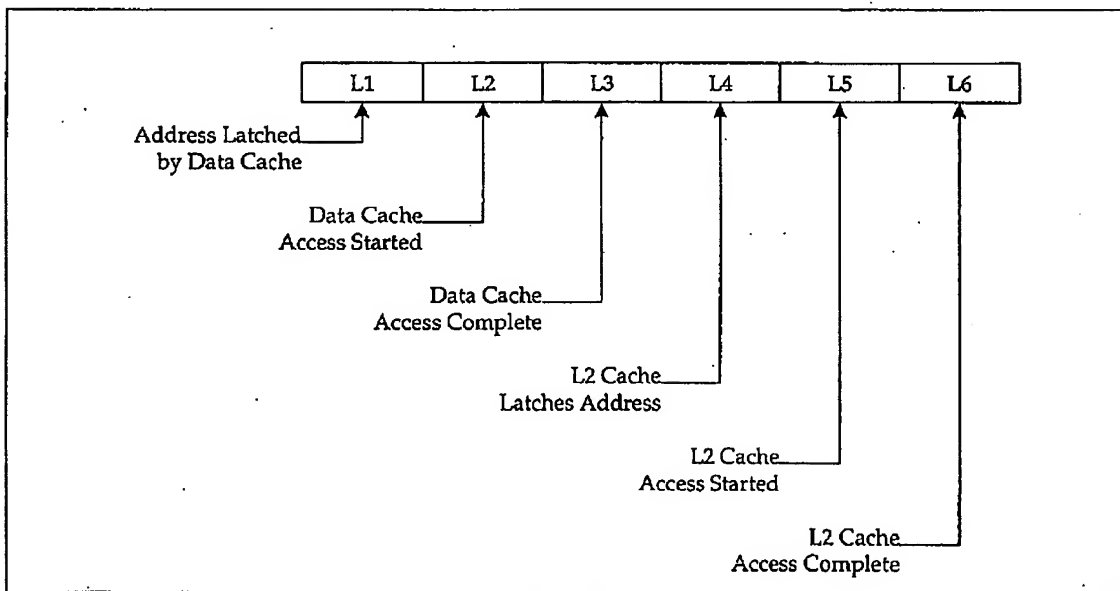


Figure 5-18: Load Pipeline Stages



Pentium Pro Processor System Architecture

Table 5-5: Load Pipeline Stages

Stage(s)	Description
L1	During the L1 stage, the memory address for the load is translated from the linear to the physical address by the data TLB at the front end of the data cache.
L2-L3	During the L2 and L3 stages, the L1 data cache is accessed. <ul style="list-style-type: none">• If this results in a hit on the L1 data cache, the data is supplied to the RS and the micro-op's ROB entry in the L3 stage. The micro-op then proceeds to the RR stage (see RR entry in this table).• If there is a miss on the L1 data cache, the instruction enters the L4 stage (see next entry).
L4-L6	In the event that the load misses the L1 data cache, the load proceeds to the L4, L5 and L6 stages where the L2 cache lookup is performed. <ul style="list-style-type: none">• In the event of an L2 cache hit, the data is delivered to the RS and the micro-op's ROB entry in L6 and the micro-op proceeds to the RR stage (see RR entry in this table).• If there is a miss on the L2 cache, the load is forwarded to the bus interface unit for an access to main memory and the micro-op stalls until it has fulfillment. Because the L1 and L2 caches are non-blocking, however, subsequent loads that hit on L1 or L2 can complete before the stalled load.
RR	Ready for retirement. When it has been established that there is no chance of a branch occurring earlier in the program flow, the load micro-op is marked RR in the ROB.
RT	Retirement stage. Data returned by the load is copied to the real target register from the ROB entry and the ROB entry becomes available for a new micro-op.

Handling Stores

An IA store (i.e., memory data write) instruction decodes into two micro-ops. When executed,

- one generates the address to be stored to.
- the other generates the data to be stored.

Chapter 5: The Fetch, Decode, Execute Engine

They can both be dispatched and executed simultaneously because the processor implements two separate execution units for handling stores:

- Store address unit generates the address to be stored to.
- Store data unit generates the data to be written.

Relative to other stores, the processor always performs stores in strict program order. A store cannot be executed until all earlier program stores have been performed. This is necessary in order to ensure proper operation of memory devices that are sensitive to the order in which information is delivered to them. In addition, stores are never speculatively executed. All upstream conditional branches must have been resolved before stores are executed.

When executed, all stores are handled in the following manner:

- If the store is within UC memory, it is posted in the posted-write buffer and, when the posted writes are performed on the bus, will be performed in strict program order relative to other stores. For a description of UC memory, refer to "Uncacheable (UC) Memory" on page 124.
- If the store is within WC memory, it is posted in a WC (write-combining) buffer to be written to memory later. When the WC buffers are written to memory, the write may not occur in the order specified by the program relative to other writes. For a description of WC memory, refer to "Write-Combining (WC) Memory" on page 124.
- If the store is within WT memory and it's a hit on the data or L2 caches, the line is updated and the write is also posted in the posted-write buffer. If the store is a cache miss, it has no effect on the caches, but is posted in the posted-write buffer. When the posted writes are performed on the bus, it will be performed in strict program order relative to other stores. For a description of WT memory, refer to "Write-Through (WT) Memory" on page 126.
- If the store is in WP memory, it has no effect on the caches, but is posted in the posted-write buffer. When the posted writes are performed on the bus, it will be performed in strict program order relative to other stores. For a description of WP memory, refer to "Write-Protect (WP) Memory" on page 126.
- If the store is in WB memory and it's a hit on the data or L2 cache, the write is absorbed into the line and is not posted in either the posted-write or WC buffers. For a description of operation within WB memory, refer to "Relationship of L2 and L1 Caches" on page 147 and to "Write-Back (WB) Memory" on page 127.

The Intel documentation does not specify the depth of the posted-write buffer.

Pentium Pro Processor System Architecture

Description of Branch Prediction

486 Branch Handling

The 486 processor does not implement any form of branch prediction. The instruction decode logic does not differentiate branches from other instruction types and therefore does not alter prefetching. In other words, instruction prefetching always continues past a branch to the next sequential instruction. If the branch is taken when it is executed, the instructions in the prefetch queue and those in the D1 and D2 stages are flushed. This creates a relatively minor bubble in the instruction pipeline (minor because the pipeline has so few stages).

Pentium Branch Prediction

The Pentium processor implements dynamic branch prediction using a very simple mechanism that yields a typical 85% hit rate. As each prefetched instruction is passed into the dual instruction pipelines, the memory address it was fetched from is used to perform a lookup in the BTB (branch target buffer). The BTB is implemented as a high-speed, look-aside cache. If there's a branch and it misses the BTB, it is predicted as not taken and the prefetch path is not altered. If it hits in the BTB, the state of the BTB entry's history bits is used to determine whether the branch should be predicted as taken or not taken. When the branch is executed, its results (whether it was taken or not and, if taken, the branch target address) are used to update the BTB. If the branch is incorrectly predicted, the instructions in the two pipelines and those in the currently-active prefetch queue must be flushed. This doesn't cause a terrible performance hit because the Pentium has relatively shallow instruction pipelines. For a more detailed discussion of the Pentium's branch prediction algorithm, refer to the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).

Pentium Pro Branch Prediction

Mispredicted Branches are VERY Costly!

The Pentium Pro processor's instruction pipeline is deep (i.e., it consists of many stages). Close to the beginning of the pipeline (in the IFU2 stage; see Fig-

Chapter 5: The Fetch, Decode, Execute Engine

ure 5-19 on page 111), the address of a branch instruction is submitted to the dynamic branch prediction logic for a lookup and, if a hit, a prediction is made on whether or not the branch will be taken when the branch instruction is finally executed. The BTB only makes predictions on branches that it has seen taken previously. Based on this prediction, the branch prediction logic takes one of two actions:

- If the branch is predicted taken, the instructions that were fetched from memory locations along the fall-through path of execution are flushed from the 16-byte block of code that is currently in the IFU2 stage. The instructions currently in the prefetch streaming buffer are also flushed. The branch prediction logic provides the branch target address to the IFU1 stage and the prefetcher begins to refill the streaming buffer with instructions from the predicted path.
- If the branch is predicted as not taken, the branch prediction logic does not flush the instructions that come after the branch in the 16-byte code block currently in the IFU2 stage. It also does not flush the streaming buffer and the prefetcher continues fetching code along the fall-through path.

The branch instruction migrates through the pre-ROB pipeline stages, is placed in the ROB and, ultimately, is executed by the JEU (jump execution unit). The series of instructions from the predicted path followed along behind it in the pipeline and were also placed in the ROB. Due to the processor's out-of-order execution engine, by the time the branch is executed a number of the instructions that come after the branch in the program flow may have already completed execution.

When the branch is finally executed, all is well if the prediction was correct. However, if the prediction is incorrect:

- all instructions currently in the ROB that came after the branch must be flushed from the ROB (along with their execution results if any of them had already been executed).
- all of the instructions that come after the branch that are currently in the RS or are currently being executed must be flushed
- all instructions in the earlier pipeline stages must be flushed
- the streaming buffer must be flushed.
- the prefetcher must then issue a request for code starting at the correct address. This is supplied by the BTB which always keeps a record of the branch target address as well as the fall through address.

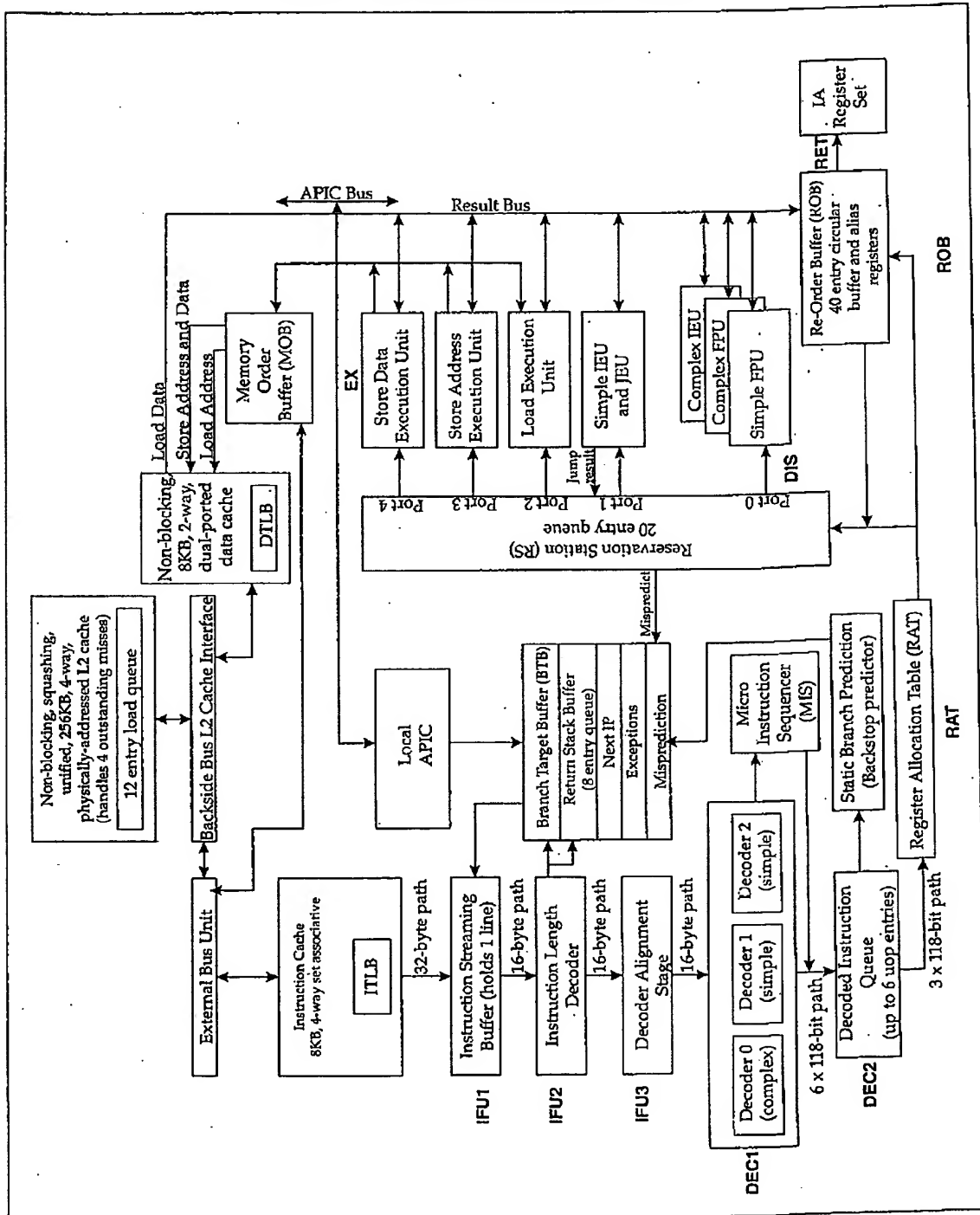
Because of the large performance hit that results from a mispredicted branch, the designers of the processor incorporate two forms of branch prediction:

Pentium Pro Processor System Architecture

dynamic and static branch prediction, and implemented a more robust dynamic branch prediction algorithm than that used in the Pentium processor. The sections that follow describe the static and dynamic branch prediction mechanisms.

Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-19: Fetch/Decode/Execute Engine



Pentium Pro Processor System Architecture

Dynamic Branch Prediction

General. Refer to Figure 5-19 on page 111. The Pentium Pro processor's BTB implements a two-level, adaptive dynamic branch prediction algorithm. *Please note that the following initial discussion temporarily ignores the static branch prediction mechanism.* In the IFU2 pipeline stage, the address of any conditional branch instruction (up to four simultaneously) that resides within the 16-byte code block is submitted to the BTB for a lookup (in other words, the BTB can make up to four branch predictions simultaneously). The first time that a branch is seen by the BTB, it results in a BTB miss (the BTB has no history on how it will execute). When the branch finally arrives at the JEU (jump execution unit) and is executed, an entry is made in the BTB. The entry records:

- the address that the branch was fetched from.
- whether or not the branch was taken and, if so, the branch target address is recorded in the BTB entry.

The size and organization of the BTB is processor design-specific. The current implementations have a 4-way set-associate BTB with a total of 512 entries (double the Pentium BTB's size).

Yeh's Prediction Algorithm. Unlike the Pentium's BTB, which uses a simple counting mechanism (a variant on the Smith algorithm, a 2-bit counter that increments each time the branch is taken and decrements each time it's not; either of the higher two values indicate a taken prediction, while the lower two values indicate a not taken prediction), the Pentium Pro processor's BTB is capable of recognizing behavior patterns such as taken, taken, not taken. The algorithm used is referred to as Yeh's algorithm and is a two-level, adaptive algorithm. Each BTB entry uses 4-bits to maintain history on the branch's behavior the last four times that it was executed. There are many ways in which this algorithm can be implemented, but Intel has declined to describe the Pentium Pro's implementation. For additional information on Yeh's algorithm, refer to a very good article in the 3/27/95 issue of Microprocessor Report (Volume 9, Number 4). It is estimated that this algorithm can achieve accuracy of approximately 90-95% on SPECint92 and better on SPECfp92.

Return Stack Buffer (RSB)

The programmer calls a subroutine by executing a call instruction, explicitly citing the address to jump to. In other words, a call instruction is an unconditional branch and is therefore always correctly predicted as taken. When the call is executed, the address of the instruction that immediately follows the call is automatically pushed onto the stack by the processor. The programmer always

Chapter 5: The Fetch, Decode, Execute Engine

ends the called routine with a return instruction. The return instruction pops the previously-pushed address off the stack and jumps to the instruction it points to in order to resume execution of the program that called the routine. In other words, the return instruction is also an unconditional branch and can always be predicted as taken. While this is true, it doesn't address the question of where it will branch to.

Previous x86 processor implementations kept no record of the address pushed onto the stack and therefore, although it could be predicted that the branch is taken, it could not "remember" where the branch would be taken to.

Each time that a return instruction is seen, the Pentium Pro processor performs a lookup in a small cache of return instructions (the return stack buffer, or RSB) that it has previously seen executed. The first time the instruction is seen, it results in a miss on the RSB. When the return instruction is executed and the return address is popped from the stack, it is recorded in the RSB for future lookups. The next time that the instruction is seen in the IFU2 stage, it results in a hit on the RSB and the RSB supplies the branch target address to the prefetcher. In cases where the called routine does not alter the pointer that was pushed onto the stack by the call instruction, the return will always be predicted correctly. However, if a routine changes the pointer dynamically, the RSB won't be correct. The Intel documentation is unclear as to the size of the RSB, but the author has seen sizes of 4 and 8 entries quoted.

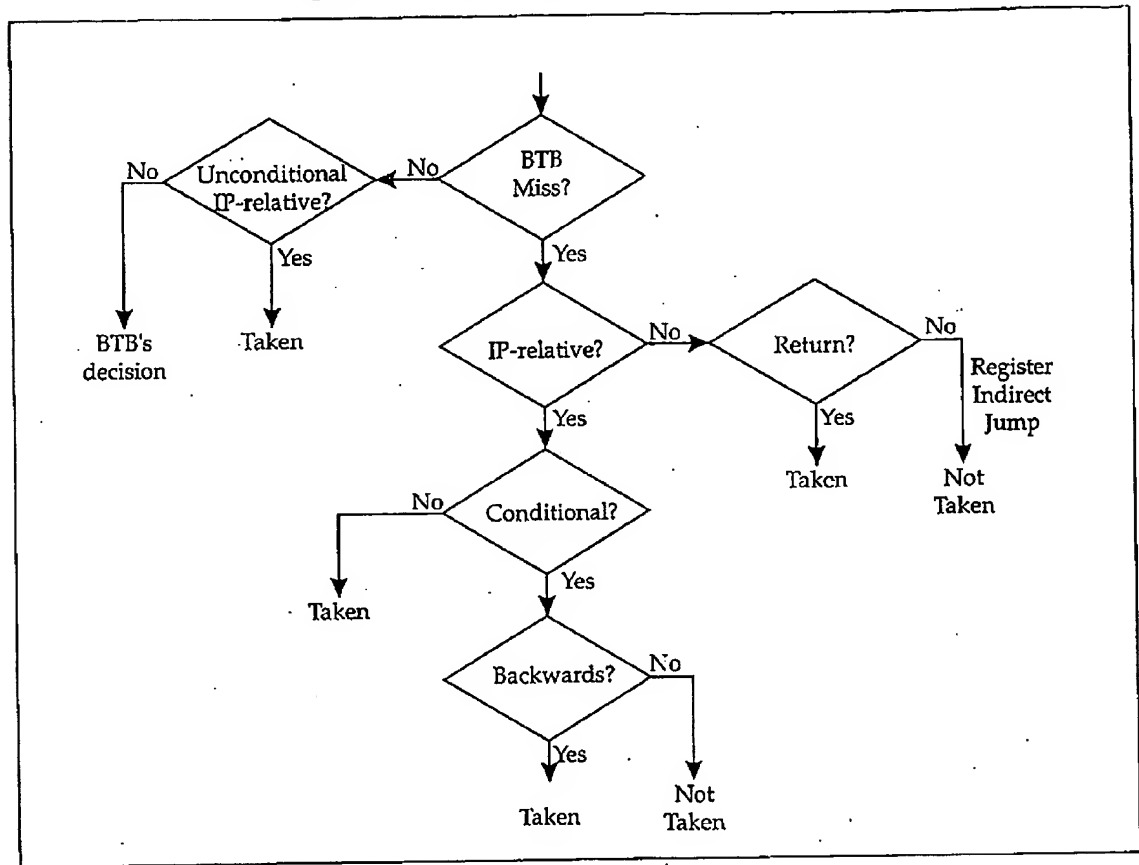
Static Branch Prediction

The static branch predictor is also referred to as the backstop mechanism because it provides a backup to the dynamic branch prediction logic. Branches are submitted to the static branch prediction logic in the DEC2 stage (see Figure 5-21 on page 115). The static branch prediction decision tree is illustrated in Figure 5-20 on page 114.

For the most part, the static branch predictor handles branches that miss the BTB. However, in the case of an unconditional IP-relative branch, the static branch predictor always forces the prediction to the taken state (and updates the BTB if there was a disagreement).

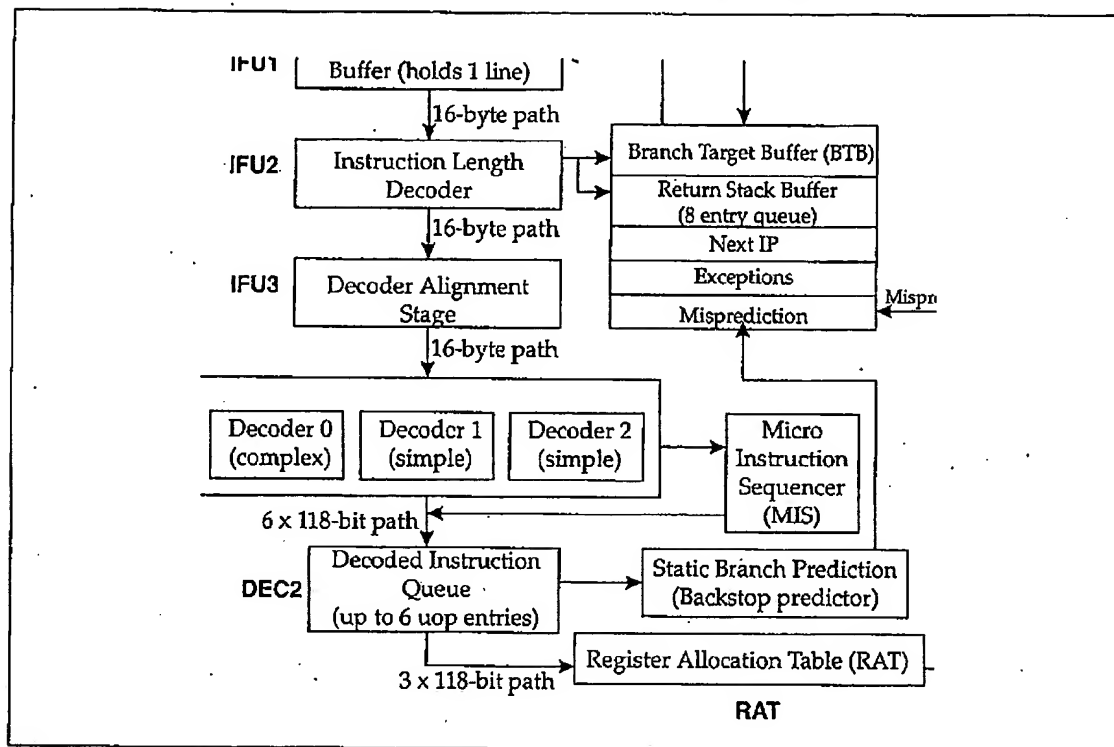
Pentium Pro Processor System Architecture

Figure 5-20: Static Branch Prediction Algorithm



Chapter 5: The Fetch, Decode, Execute Engine

Figure 5-21: Static Branch Prediction Logic



Code Optimization

General

This section highlights some of the code optimizations recommended by Intel to yield the best processor performance.

Reduce Number of Branches

As described in "Pentium Pro Branch Prediction" on page 108, mispredicted branches cause a severe performance penalty. Wherever possible, eliminate branches. One excellent way to do this is by using the CMOV and FCMOV instructions (see "Conditional Move (CMOV) Eliminates Branches" on page 367 and "Conditional FP Move (FCMOV) Eliminates Branches" on page 367).

Pentium Pro Processor System Architecture

Follow Static Branch Prediction Algorithm

In order to optimize performance for branches that miss the BTB, craft branches to take advantage of the static branch prediction mechanism (see "Static Branch Prediction" on page 113).

Identify and Improve Unpredictable Branches

Indirect branches, such as switch statements, computed GOTOs, or calls through pointers can branch to an arbitrary number of target addresses. When the branch goes to a specific target address a large amount of the time, the dynamic branch predictor will handle it quite well. However, if the target address is fairly random, the branch prediction may not have a very good rate of success. It would be better to replace the switch, etc., with a set of conditional branches that the branch prediction logic can handle well.

Don't Intermingle Code and Data

Don't intermingle code and data within the same cache line. The line can then end up residing in both the code and data caches. If the data is modified, the processor treats this as if it is self-modifying code (see "Self-Modifying Code and Self-Snooping" on page 173), resulting in poor performance.

Align Data

Data objects should be placed in memory aligned within a dword wherever possible. A data object (consisting of 2 or 4 bytes) that straddles a dword boundary may end up causing a lengthy delay to read or write the object. The worst-case scenario would be a data object that straddles a page (i.e., a 4KB) address boundary and causes two page fault exceptions (because neither page is currently in memory). This would result in a stall of the current program until both pages had been read from mass storage into memory.

Avoid Serializing Instructions

Refer to "CPLD is a Serializing Instruction" on page 366. Serializing instructions such as CPLD restrain the processor from performing speculative code execution and has a serious impact on program execution. Sometimes they must be used to achieve a specific goal, but they should be used as sparingly as

Chapter 5: The Fetch, Decode, Execute Engine

possible. The following instructions cause execution serialization:

- Privileged instructions—MOV to control register, MOV to debug register, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, and LTR.
- Non-privileged instructions—CUID, IRET, and RSM.

Where Possible, Do Context Switches in Software

When the processor hardware is used to perform a task switch (by jumping through a task gate descriptor), the bulk of the processor's register set is automatically saved to the TSS (task state segment) for the program being suspended, and the register set is then reloaded with the register image from the TSS associated with the program being started (or resumed). This save and reload takes quite a bit of time. It may be possible to suspend the current task by saving a smaller subset of the register set (rather than the entire register set), and it may be possible to start (or resume) the new task by loading just a few of the registers. In this case, performance would be aided if the programmer performs the state save and reload using software.

Eliminate Partial Stalls: Small Write Followed by Full-Register Read

If a full register (EAX, EBX, ECX, or EDX) is read (e.g.—`mov ebx, eax`) after part of the register is written to (e.g.—`mov al, 5`), the processor experiences a stall of at least 7 clocks (and maybe much longer). The micro-op that reads from the full register is stalled until the partial write is retired, writing the data to the subset of the real IA register. Only then is the value read from the full, real IA register. In addition, none of the micro-ops that follow the full register read micro-op will be executed until the full register read completes.

Since 16-bit code performs partial register writes a lot, the processor suffers poor performance when executing 16-bit code.

Data Segment Register Changes Serialize Execution

16-bit code frequently changes the contents of data segment registers (DS, ES, FS, and GS). Unlike the processor's general-purpose registers, the data segment registers are currently not aliased. A write to the register immediately changes the value in the real, IA data segment register. Micro-ops that reside down-

Pentium Pro Processor System Architecture

stream from the micro-op that changes the register may perform accesses (loads or stores) within the affected data segment. If the processor were permitted to speculatively execute instructions beyond the one that changes the data segment register before that micro-op had completed, they would be using the old, stale contents of the segment register and would address the wrong location.

For this reason, the processor will not execute any instructions beyond the segment register load until the load has completed. The processor's performance degrades because it is restrained from out-of-order execution. *Since 16-bit code changes the contents of the data segment registers a lot, the processor suffers poor performance when executing 16-bit code.*

More than likely, Intel will fix this problem in subsequent versions of the Pentium Pro processor by aliasing the data segment registers as is already done for the general-purpose registers.

6

Rules of Conduct

The Previous Chapter

The previous chapter provided a detailed description the processor logic responsible for instruction fetch, decode, and execution.

This Chapter

In preparation for the next chapter's discussion of the processor's caches, this chapter introduces the register set that tells the processor its rules of conduct within various areas of memory. The Memory Type and Range Registers, or MTRRs, must be programmed after startup to define the various regions of memory within the 64GB's of memory space and how the processor core and caches must behave when performing accesses within each region. A detailed description of the MTRRs may be found in the appendix entitled "The MTRR Registers" on page 505.

The Next Chapter

The next chapter provides a detailed description of the processor's L2, L1 data, and L1 code caches.

The Problem

General

The overall memory space that the processor may read and write can be populated with many different types of devices. The operational characteristics of a device must dictate how the processor (and its caches) behave when performing accesses within the memory range assigned to a device. Not following the correct "rules of conduct" when performing accesses within the device's range can lead to a confused device and improper operation.

Pentium Pro Processor System Architecture

A Memory-Mapped IO Example

As an example, if a memory range is populated by one or more memory-mapped IO devices and the processor were to perform speculative reads from or were to cache from that memory range, the memory-mapped IO devices would be hopelessly confused. The programmer might perform a read of a single location to read a device's status port and, without the programmer's knowledge, the processor performs a cache line fill to obtain the requested byte plus the other 31 locations that occupy the same line. The read of the other 31 locations might read data from FIFO buffers and status from other status registers within the same or other memory-mapped IO devices. The device(s), thinking that the data and/or status had been read, would move new data into the FIFO locations, and might clear status bits that had been set in the status ports. In other words, the device(s) would be left in a very different state than the programmer expects and would no longer function as expected.

Pentium Solution

The only Pentium mechanism available to the programmer to define the rules of conduct within a specific memory range are the PCD and PWT bits in a page table entry. This gives the OS programmer the ability to define the rules of conduct with 4KB granularity. Unfortunately, the OS is typically not platform-specific and doesn't necessarily have an accurate view of the denizens that occupy various areas of memory space. This means that although the OS may have informed the processor (via the two bits in the page table entry) that the memory area being accessed is cacheable, the platform-specific hardware may have been configured (by the BIOS code) with a more accurate view of the map and who lives where.

When the Pentium initiates a memory access, it projects its rules of conduct on its PCD and PWT output pins. If it considers the memory region cacheable, it asserts its CACHE# output to request the entire line. Before proceeding with the data transfer, however, it first samples its KEN# input to see if the chipset has examined the memory address and agrees that is cacheable. If the processor is attempting a cache line fill, but KEN# is sampled deasserted, too bad. The chipset will only return the data indicated by the byte enables, not the whole line.

Chapter 6: Rules of Conduct

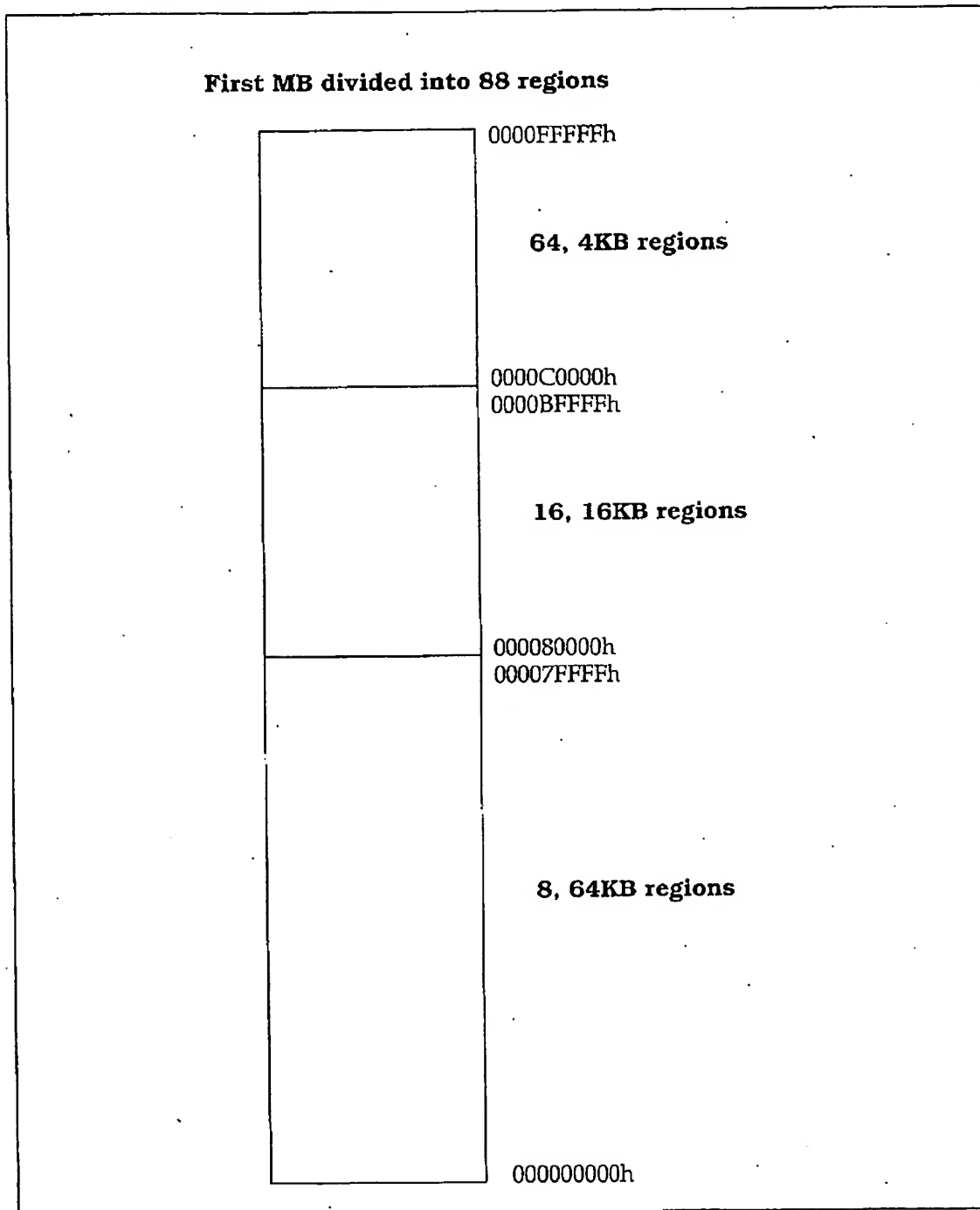
Pentium Pro Solution

The Pentium Pro processor includes a set of registers that the platform-specific software (e.g., the BIOS or the OS HAL) sets up to define the rules of conduct throughout the 4GB range. The Memory Type and Range Registers, or MTRRs, are implemented as model-specific registers and are accessed using the RDMSR and WRMSR instructions (see the chapter entitled "Instruction Set Enhancements" on page 359).

The MTRR register set is divided into fixed-range and variable-range registers. The fixed-range registers define the processor's rules of conduct within the first MB of memory. If present and enabled, the fixed-range MTRRs divide the first MB into 88 regions, with a memory type assigned to each (illustrated in Figure 6-1 on page 122). Each of the variable-range registers can be programmed with a start and end address and the rules of conduct within the programmer-defined range. A detailed description of the MTRRs can be found in the appendix entitled "The MTRR Registers" on page 505.

Pentium Pro Processor System Architecture

Figure 6-1: Fixed-Range MTRRs Define Rules of Conduct within First MB of Memory Space

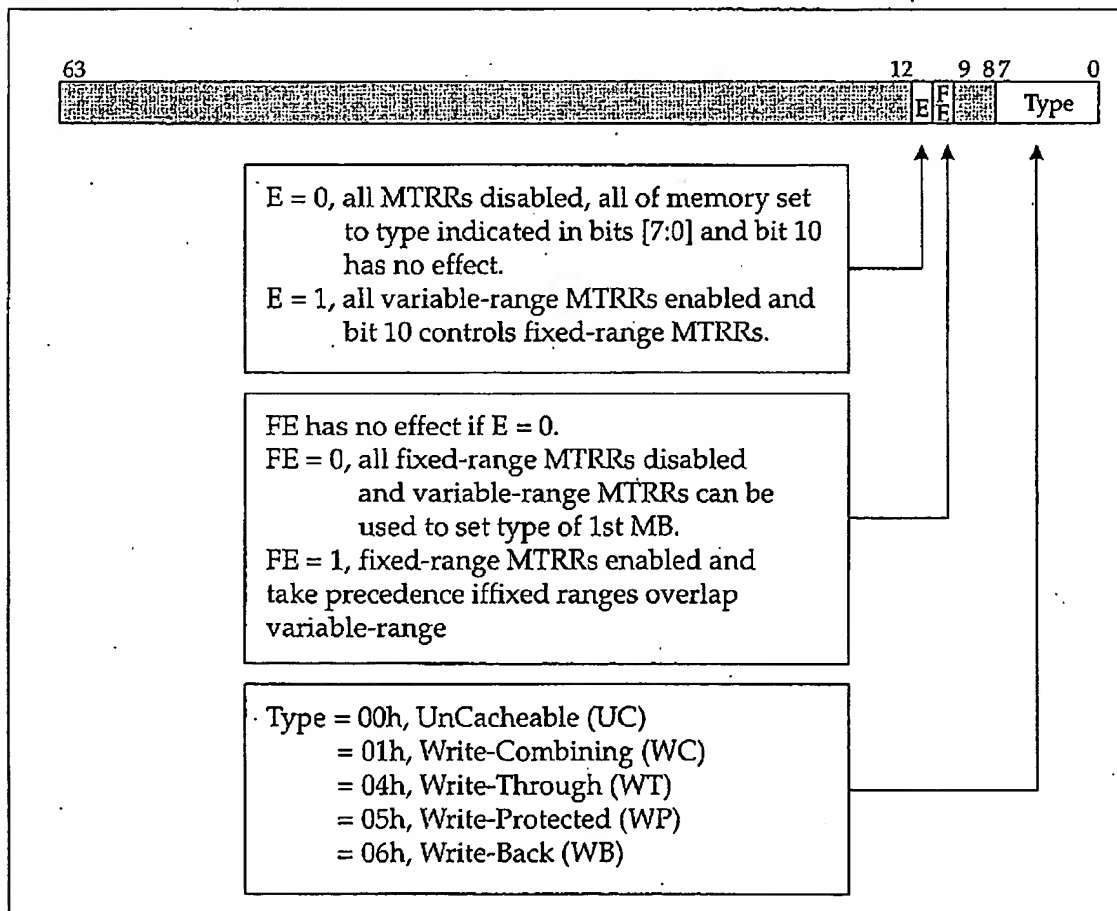


Chapter 6: Rules of Conduct

State of the MTRRs after Reset

Reset disables all of the MTRRs by clearing the MTRRdefType register (see Figure 6-2 on page 123), setting all of memory space to the uncacheable (UC) memory type. After the processor starts executing the POST, the programmer can change the default rules of conduct by changing the value in the register's TYPE field. The MTRRdefType register is implemented as a Model-Specific Register, or MSR, and is accessed using the RDMSR and WRMSR instructions (see the chapter entitled "Instruction Set Enhancements" on page 359). Once the MTRRs are enabled, the rules of conduct (i.e., the memory type) specified in the MTRRdefType register are used for any accesses within ranges not covered by the MTRRs.

Figure 6-2: MTRRdefType Register



Pentium Pro Processor System Architecture

Memory Types

Using the MTRRdefType register or the individual fixed-length and/or variable range MTRRs, the programmer can define each memory range as one of five types (note that they are listed in order of aggressiveness: UC yields very low performance while WB yields the best overall performance):

- Uncacheable, or UC.
- Write-Combining, or WC.
- Write-Through, or WT.
- Write-Protect, or WP.
- Write-Back, or WB.

The rules of conduct within the five types of memory areas are defined in the sections that follow. *The Pentium Pro processor never performs speculative writes (because there is no way to “undo” the write if it is later discovered that the write should not have been done).*

Uncacheable (UC) Memory

The rules that the processor follows when performing memory reads and writes in a memory range designated as UC are as follows:

- Cache lookups are not performed.
- Read requests are not turned into line reads from memory. They are performed as is. The data returned is routed directly to the requestor and is not placed in any cache.
- Memory writes are first posted in the processor’s posted-write buffer and later performed on the bus in original program order.
- Speculative reads are not performed.

In other words, the processor is very well-behaved within a UC memory. For this reason, the UC type is well-suited to memory regions populated by memory-mapped IO devices. On the negative side, accesses within UC memory yield low performance due to lack of caching and speculative read constraining.

Write-Combining (WC) Memory

Note that the programmer must check the WC bit in the MTRRcap (see “The MTRR Registers” on page 505) register to determine if the WC memory type is supported. The

Chapter 6: Rules of Conduct

rules that the processor follows when performing memory reads and writes in a memory range designated as WC are as follows:

- Cache lookups are not performed.
- Read requests are not turned into line reads from memory. They are performed as is. The data returned is routed directly to the requestor and is not placed in any cache.
- Speculative reads permitted.
- When one or more bytes are to be written to memory, a 32-byte Write Combining Buffer (WCB) memorizes the 32-byte aligned start address of the area the bytes are to be written to as well as the bytes to be written.
- The contents of the WCB are written to memory under the following conditions:
 - If any additional writes within WC memory are performed to a different 32-byte area and all of the WCBs are in use, the processor will flush one of the WCBs to memory. The empty WCB will then be available to record the new 32-byte aligned start address and the bytes to be written to that 32-byte block.
 - Execution of a serializing instruction (e.g., CUID, IRET, RSM), an IO instruction, or a locked operation causes the processor to flush all write buffers to memory before proceeding to the next instruction.

When flushing a WCB to memory, the processor will take one of two courses of action:

- If the WCB is full (i.e., all 32 bytes have been posted), the processor will perform a single 32-byte write transaction to write the entire line to memory.
- If all bytes in the WCB are not valid, the processor will perform the appropriate number of quadword or partial-quadword write transactions to write the updates to memory. As an example, if there are five bytes to be written to the first quadword and three to be written to the third quadword of the 32-byte block, the processor will perform two separate write transactions: one using the start address of the first quadword with five byte enables asserted; and the other using the start address of the third quadword with three byte enables asserted.

It should be noted that when these writes appear on the bus, the byte enables may or may not be contiguous, depending on the bytes to be written within the quadword. The WC memory type is useful for linear video frame buffers.

Pentium Pro Processor System Architecture

Write-Through (WT) Memory

The rules that the processor follows when performing memory reads and writes in a memory range designated as WT are as follows:

- Cache lookups are performed.
- On a cache read miss, the entire line is fetched from memory and placed in the cache.
- On a cache read hit, the requested data is supplied by the cache and a bus transaction does not take place.
- On a cache write miss, the data is posted in the processor's posted-write buffer to be written to memory later.
- On a cache write hit, the cache line is updated (but the line is not marked modified) and the data is posted in the processor's posted-write buffer to be written to memory later.
- Lines are never marked modified. They may only be in the S or I states.
- Speculative reads allowed.
- A write updates the L1 data cache, but invalidates the L2 and L1 code caches on a hit (see the section on self-modifying code in the chapter entitled "The Processor Caches" on page 133).

Write-Protect (WP) Memory

The rules that the processor follows when performing memory reads and writes in a memory range designated as WP are as follows:

- Cache lookups are performed.
- On a cache read miss, the entire line is fetched from memory and placed in the cache.
- On a cache read hit, the requested data is supplied by the cache and a bus transaction does not take place.
- On a cache write miss, the data is posted in the processor's posted-write buffer to be written to memory later.
- On a cache write hit, the cache line is *not* updated and the data is posted in the processor's posted-write buffer to be written to memory later.
- Lines are never marked modified. They may only be in the S or I states.

The WP memory type is useful for shadow RAM that contains ROM code. It may be cached for good performance, but cannot be changed in the cache (thereby simulating ROM memory). It should be noted that writes, although not

Chapter 6: Rules of Conduct

performed in the cache, are performed to main memory. It is the memory controller's responsibility to ignore the memory write (note that the memory controller can examine the ATTRIB[7:0]# bits in the transaction request to determine the memory type).

Write-Back (WB) Memory

The rules that the processor follows when performing memory reads and writes in a memory range designated as WB are as follows:

- Cache lookups are performed.
- On a read cache miss, the entire line is fetched from memory and placed in the cache (in the E or S state).
- On a read cache hit, the requested data is supplied by the cache and a bus transaction does not take place.
- On a cache write miss, the entire line is read from memory using the read and invalidate transaction type. This is referred to as an *allocate-on-write miss policy*. Any other cache that has a snoop hit on a line in the E or S state must invalidate its copy (i.e., E->I, or S->I). Any other cache with a snoop hit on a line in the M state must source the 32-byte modified line to the requesting processor and to memory and invalidate its copy (i.e., M->I). When the requesting processor receives the line, it places it into the cache, writes into it and marks it as M.
- On a cache write hit, the cache line is updated. If the line was in the E state, it is changed to the M state (E->M). If it was in the M state, it stays in the M state (M->M). In either case, no bus transaction takes place and main memory is not updated.
- Speculative reads permitted.

The WB memory type yields the best overall performance. Reads and writes can be serviced solely in the cache without performing bus transactions, so bus traffic is greatly diminished. Ideally, most of main memory should be designated as the WB memory type.

Rules as Defined by MTRRs

Assuming that the MTRRs are enabled and have been set up by the programmer, the processor interrogates the MTRRs for any memory access to determine its rules of conduct. The determination is made as indicated in Table 6-1 on page 128.

Pentium Pro Processor System Architecture

Table 6-1: Memory Type Determination Using MTRRs

Scenario	Resulting Memory Type
Memory address within first MB, fixed-range MTRRs present and enabled, and address not within range defined by any of the variable-length MTRRS.	Type defined by the fixed-length MTRR for the range.
Address within first MB, fixed-length MTRRs disabled (or not present), and address not within a range defined by variable-length MTRRs.	Type defined by TYPE field in the MTRRdefType register.
Address within first MB, fixed-length MTRRs disabled (or not present), and address within a range defined by one variable-length MTRR.	Type defined by the variable-length MTRR for the range.
Address within first MB, fixed-length MTRRs disabled (or not present), and address within a range defined by more than one variable-length MTRR.	<ul style="list-style-type: none"> • If all of the respective variable-length MTRRs define the range as UC, then the type is UC. • If the respective variable-length MTRRs define the range as UC and WB, then the type is UC. • If the respective variable-length MTRRs define the range as other than UC and WB, then the behavior of the processor is undefined.
Address not within first MB, but not within a range defined by any of the variable-length MTRRs.	Type defined by TYPE field in the MTRRdefType register.
Address not within first MB and within a range defined by one of the variable-length MTRRs.	Type defined by TYPE field in a variable-range MTRR register.

Chapter 6: Rules of Conduct

Table 6-1: Memory Type Determination Using MTRRs (Continued)

Scenario	Resulting Memory Type
Address not within first MB and within ranges defined by more than one variable-length MTRRs.	<ul style="list-style-type: none"> • If all of the respective variable-length MTRRs define the range as UC, then the type is UC. • If the respective variable-length MTRRs define the range as UC and WB, then the type is UC. • If the respective variable-length MTRRs define the range as other than UC and WB, then the behavior of the processor is undefined.

Rules of Conduct Provided in Bus Transaction

Whenever the processor performs a memory read or write bus transaction, it outputs the memory type on the ATTRIB[7:0]# signals as part of the transaction request. In this manner, the memory controller and an L3 cache (if present) are also informed of the rules of conduct within the addressed memory area.

MTRRs and Paging: When Worlds Collide

Table 6-2 on page 129 and Table 6-3 on page 130 define the resulting rules of conduct when the MTRRs and page table entries agree or disagree on the memory type. Basically, when the target address of the memory access is covered by both a page table entry and an MTRR, the resulting memory type is the more conservative (i.e., the safer) of the two.

Table 6-2: Basic Relationship of the MTRRs to Paging

MTRR State	Paging State	Resulting Rules of Conduct
MTRRs disabled (MTRRdefType register = 0).	Paging disabled (CR0[PG] = 0).	MTRRdefType register's TYPE field defines default memory type.

Pentium Pro Processor System Architecture

Table 6-2: Basic Relationship of the MTRRs to Paging (Continued)

MTRR State	Paging State	Resulting Rules of Conduct
MTRRs enabled.	Paging disabled.	MTRR for target memory range defines memory type.
MTRRs disabled.	Paging enabled (CR0[PG] = 1).	PCD and PWT bits in selected page table entry define memory type as uncacheable, cacheable write-through, or cacheable write-back.
MTRRs enabled.	Paging enabled.	and target address is covered by both...see Table 6-3 on page 130.

Table 6-3: Type if Paging and MTRRs enabled and Address Covered by Both

MTRR Type	PCD	PWT	Resulting Memory Type
UC	x	x	UC
WC	0	0	WC
	0	1	WC
	1	0	WC
	1	1	UC
WT	0	x	WT
	1	x	UC

Chapter 6: Rules of Conduct

Table 6-3: Type if Paging and MTRRs enabled and Address Covered by Both (Continued)

MTRR Type	PCD	PWT	Resulting Memory Type
WP	0	0	WP
	0	1	WP
	1	0	UC
	1	1	UC
WB	0	0	WB
	0	1	WT
	1	x	UC

Detailed Description of the MTRRs

A detailed description of the MTRRs can be found in the appendix entitled "The MTRR Registers" on page 505.

7

The Processor Caches

The Previous Chapter

In preparation for this chapter's discussion of the processor's caches, the previous chapter introduced the register set that tells the processor its rules of conduct within various areas of memory. The Memory Type and Range Registers, or MTRRs, must be programmed after startup to define the various regions of memory within the 64GB's of memory space and how the processor core and caches must behave when performing accesses within each region.

This Chapter

This chapter provides a detailed description of the processor's L1 data and code caches, as well as its unified L2 cache. This includes a discussion of self-modifying code and toggle mode transfer order during the transfer of a cache line on the bus.

The Next Chapter

This chapter concludes Section One of the hardware part of the book—the discussion of the processor's internal operation. The next chapter starts Section Two, the bus section, and introduces the electrical characteristics of the Pentium Pro processor's bus.

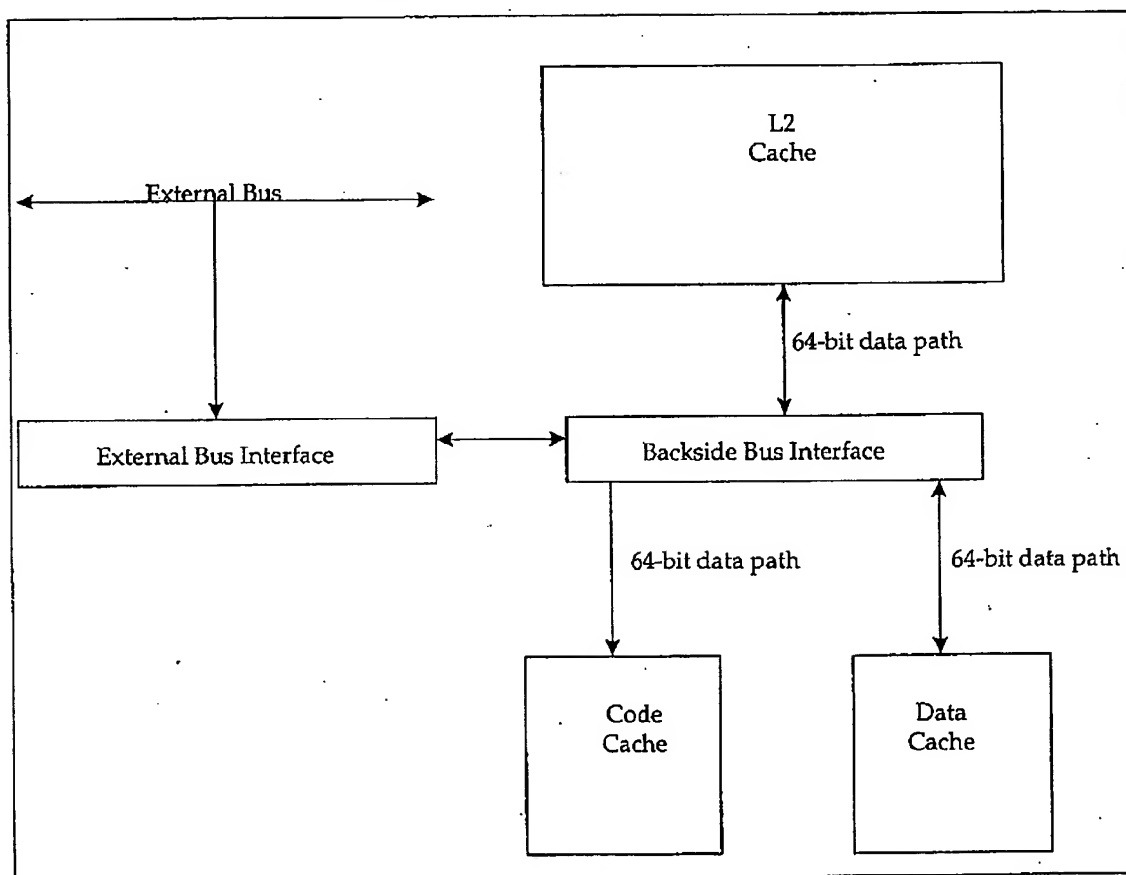
Cache Overview

It should be noted that the size and structure of the processor's L2 cache and L1 data and code caches are processor implementation-specific.

Pentium Pro Processor System Architecture

Figure 7-1 on page 134 provides an overview of the processor's caches. The L1 code cache services requests for instructions generated by the instruction prefetcher (the prefetcher is the only unit that accesses the code cache and it only reads from it, so the code cache is read-only), while the L1 data cache services memory data read and write requests generated by the processor's execution units when they are executing any instruction that requires a memory data access. The unified L2 cache resides on a dedicated bus referred to as the backside bus. It services misses on the L1 caches, and, in the event of an L2 miss, it issues a transaction request to the external bus unit to obtain the requested code or data line from external memory. The information is placed in the L2 cache and is also forwarded to the appropriate L1 cache for storage.

Figure 7-1: Processor Cache Overview



Chapter 7: The Processor Caches

Introduction to Data Cache Features

The initial implementation of the data cache is 8KB, 2-way set-associative, implementing all four of the MESI cache states. It services memory data read and write requests initiated by the processor execution units and has the following features:

- **ECC protected.** Each entry in the directories and the data storage ways are ECC (Error Correcting Code) protected. For more information, see "ECC Error Handling" on page 176.
- The L1 data cache handles memory write misses in areas of memory designated as WB (writeback) by performing an **allocate-on-write** operation. For more information, see "Write Miss On L1 Data Cache" on page 151.
- **Non-blocking**—a miss goes to the bus, but does not inhibit lookups for subsequent memory data accesses submitted by the execution units.
- Can handle up to 4 misses at once. When four L2 cache accesses are already in progress and additional misses occur on the L1 data cache, the data cache cannot forward any additional access requests that miss the data cache to the L2 until one of the outstanding requests completes.
- The data cache is **pipelined**—on back-to-back execution unit data accesses that hit on the data cache, it can produce one result per processor clock.
- It can handle up to four pipelined lookups simultaneously.
- Implements two data ports and can service one load request, or a simultaneous load and store (as long as they target different lines or different halves of the same line).
- Implements a snoop port to handle snooping of transactions generated by other initiators.

For detail on the data cache, refer to the section entitled "L1 Data Cache" on page 143.

Introduction to Code Cache Features

The initial implementation of the code cache is an 8KB, 4-way set-associative cache that only implements two of the MESI cache states: S and I (because the code cache is only read by the processor core, never written to). For detail on the code cache, refer to the section entitled "L1 Code Cache" on page 137.

Pentium Pro Processor System Architecture

Introduction to L2 Cache Features

The current processors implement a unified L2 cache that services misses on the L1 code and data caches. Future versions of the processor may or may not include the L2 cache (e.g., the Klamath implementation does not include the L2). Unified refers to the fact that the L2 cache keeps copies of both code and data lines for the two L1 caches. The initial implementations are either 256KB or 512KB in size and have a 4-way set-associative structure. The L2 cache has the following characteristics:

- **ECC protected.** Each entry in the directories and the data storage ways are ECC (Error Correcting Code) protected. For more information, see "ECC Error Handling" on page 176.
- The L2 cache is **non-blocking**—a miss goes to the bus, but does not inhibit lookups for subsequent L1 cache misses.
- When a miss has already occurred for a line and the line is in the process of being fetched from memory, any subsequent misses for the same line do not generate additional memory reads (i.e., the redundant reads are squashed.)
- **Can handle up to 4 misses at once.** When four external memory read line transactions caused by L2 misses are already in progress and additional misses occur on the L1 caches, L2 hits supply data to the L1 caches while L2 misses are stored in a queue that can hold up to 12 lookup requests. As each memory read completes, the additional requests are popped off the queue one at a time and the lookup is performed.
- The L2 cache is **pipelined**—on back-to-back L1 cache misses that hit on the L2 cache, it can complete one lookup per processor clock.
- It can handle up to four pipelined lookups simultaneously.

For details on the L2 cache, refer to the section entitled "Unified L2 Cache" on page 162.

Introduction to Snooping

In order to ensure that this processor's core and other processors that initiate memory bus transactions are always dealing with the freshest copy of the information, the processor always snoops memory bus transactions initiated by other processors. Snooping is performed as follows:

1. Latch all transaction requests generated by other initiators.

Chapter 7: The Processor Caches

2. If it is a memory transaction, present the latched memory address to the three internal caches for a lookup.
3. If the targeted line isn't present in any of the caches, indicate a miss as the snoop result in the snoop phase of the transaction.
4. If the targeted line is present in one or more of the processor's caches but hasn't been modified since it was read from memory (in other words, the line is in the E or S state), indicate a snoop hit (unless the snooped transaction is a write or a read and invalidate; in that case, a cache miss is indicated and the line is invalidated) to the initiator of the transaction in the snoop phase of the transaction. Additional actions that may be necessary are defined later in this chapter.
5. If the targeted line is in the processor's data or L2 cache in the modified state (i.e., it has been modified by this processor since it was read from memory and the update hasn't been written to memory), indicate a snoop hit on a modified line to the initiator of the transaction in the snoop phase of the transaction. Additional actions that may be necessary are defined later in this chapter.

In addition, other processors snoop memory accesses that this processor generates on the bus and report back the snoop result to the initiating processor. Additional information can be found later in this chapter.

Determining Processor's Cache Sizes and Structures

The CPUID instruction may be executed with a request to return information regarding the size and organization of:

- L2 cache
- L1 data cache
- L1 code cache
- Code TLB
- Data TLBs

For detailed information on the CPUID instruction, refer to the chapter entitled "Instruction Set Enhancements" on page 359.

L1 Code Cache

As stated earlier, the size and structure of the L1 code cache (also referred to as the instruction cache or Icache) is processor implementation-specific. The initial versions of the processor implement an 8KB, 4-way, set-associative code cache

Pentium Pro Processor System Architecture

(pictured in Figure 7-2 on page 139), but as processor core speeds increase, it is likely that the cache sizes will also be increased because the faster core can process code faster. Each entry in the directories and the data storage ways are ECC (Error Correcting Code) protected. For more information, see "ECC Error Handling" on page 176.

Code Cache Uses MESI Subset: S and I

The code cache exists for only one reason: to supply requested code to the instruction prefetcher. The prefetcher issues only read requests to the code cache, so it is a read-only cache. A line stored in the code cache can only be in one of two possible states, valid or invalid, implemented as the S and I states. In other words, the code cache implements a subset of the MESI cache protocol consisting of the S and I states.

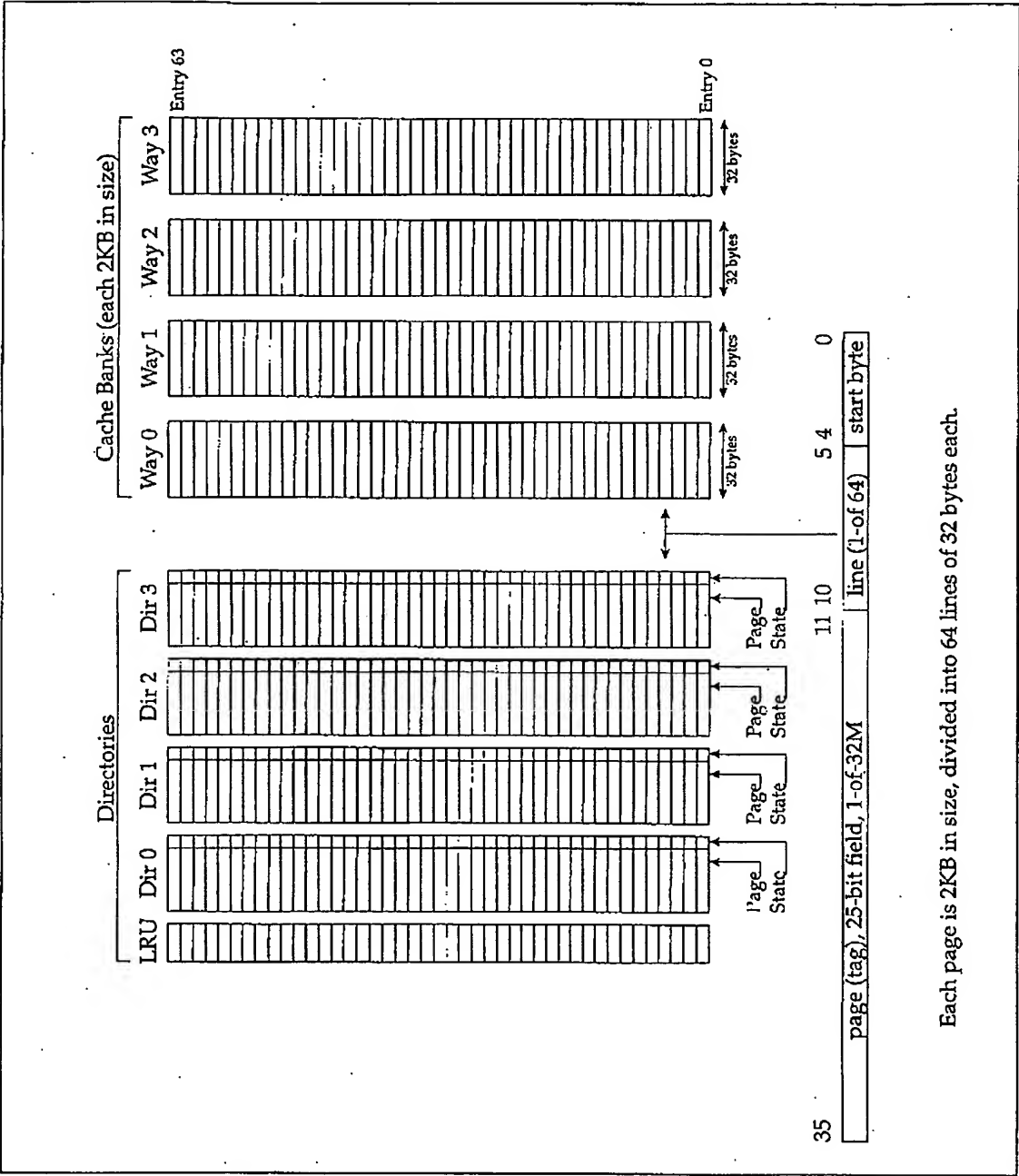
Code Cache Contains Only Raw Code

When a line of code is fetched from memory and is stored in the code cache, it consists of raw code. The designers could have chosen to prescan the code stream as it is fetched from memory and store boundary markers in the code cache to demark the boundaries between instructions within the cache line. This would preclude the need to scan the code line as it enters the instruction pipeline for decode so each of the variable-length IA instructions can be aligned with the appropriate decoder. However, this would bloat the size of the code cache. Note that the Pentium's code cache stores boundary markers.

Rather, the designers chose to store only raw code in the code cache. As each 16-byte block of code is forwarded to the IFU2 pipeline stage, instruction boundary markers are dynamically inserted in the block (in the IFU2 stage) before forwarding it to the IFU3 stage to align the instructions with the instruction decoders. For a detailed description of the pipeline stages, refer to the chapter entitled "The Fetch, Decode, Execute Engine" on page 61.

Chapter 7: The Processor Caches

Figure 7-2: The L1 Code Cache



Pentium Pro Processor System Architecture

Code Cache View of Memory Space

When performing a lookup, the code cache views memory as divided into pages equal to the size of one of its cache banks (or ways). Each of its cache ways is 2KB in size, so it views the 64GB of memory space as consisting of 32M pages, each 2KB in size. Furthermore, it views each memory page as having the same structure as one of its cache ways (i.e., a 2KB page of memory is subdivided into 64 lines, each of which is 32 bytes in size).

When a 36-bit physical memory address is submitted to the code cache by the instruction prefetcher, it is viewed as illustrated in Figure 7-2 on page 139:

- The upper 25 bits identifies the target page (1-of-32M).
- The middle 6 bits identifies the line within the page (1-of-64).
- The lower 5 bits identify the exact start address of the target instruction within the line and is not necessary to perform the lookup in the cache.

Code TLB (ITLB)

The code TLB is incorporated in the front end of the code cache, translating the linear code address produced by the prefetcher into the physical memory address before the lookup is performed. The size and organization of the code TLB is processor design-specific. The current versions of the processor have the following code TLB geometries:

- The TLB for page table entries related to 4KB code pages is 4-way set-associative with 64 entries.
- The TLB for page table entries related to 4MB code pages is 4-way set-associative with 4 entries.

Code Cache Lookup

Refer to Figure 7-2 on page 139. The target line number, contained in address bits [10:5], is used to index into the code cache directory and select a set of four entries to compare against. Each directory entry consists of a 25-bit tag field and a 1-bit state field. The cache compares the target page number to each of the selected set of four entries that are currently in the S state.

Chapter 7: The Processor Caches

Code Cache Hit

If the target page number matches the tag field in one of the entries in the S state, it is a cache hit. The code cache has a copy of the desired line from the target page. The line is in the cache way associated with the directory entry, stored in the line identified by address bits [10:5]. The code cache supplies the requested 32-byte line to the instruction streaming buffer (i.e., the instruction prefetch queue). The prefetch queue in turn forwards a 16-byte code block to the next stage of the pipeline where:

- Instruction boundary markers can be placed in it.
- If the code block contains any branches, the dynamic branch prediction logic (i.e., the BTB) attempts to predict whether or not the branch will be taken when it arrives at the execution stage.

A detailed description of the processor core may be found in the chapter entitled "The Fetch, Decode, Execute Engine" on page 61.

Code Cache Miss

On a code cache miss, the memory read request is forwarded to the L2 cache for a lookup. For a discussion of the code cache's relationship to the L2 cache, refer to "Relationship of L2 to L1 Code Cache" on page 148. For a detailed description of the L2 cache, refer to the section entitled "Unified L2 Cache" on page 162.

Code Cache LRU Algorithm: Make Room for the New Guy

When a miss occurs in the L1 code cache, the line is supplied either by the L2 cache or from external memory. In either case, the new line must be stored in the code cache. The discussion of the cache lookup described how the target line number is used as the index into the cache directory and selects a set of four entries to compare against.

Assume that all four of the selected set of entries are currently in use. In other words, the code cache has copies of the targeted line, but from four pages other than the desired page. A cache miss occurs. As stated earlier, the line will be supplied either from the L2 cache or from memory. When received, it should be obvious that the line must be stored in the code cache. It will be stored in the cache in the same relative position as it resided in within the memory page—in other words, within the same line number in one of the four cache ways. One of

Pentium Pro Processor System Architecture

the four entries in the selected set must therefore be overwritten (i.e., cast out) with the new line and the new page number must be stored in the same entry of the associated directory.

Figure 7-2 on page 139 illustrates that each set of four directory entries has an associated LRU (Least-Recently Used) bit field that is used to keep track of the least-recently used of the set of four entries. Intel does not document the LRU bit field width or usage, but it wouldn't be a big surprise if they used the same algorithm as was used for the 4-way, set-associative cache found in the 486 processor. In the 486, a 3-bit LRU field is associated with each set of four entries. LRU bit 0 indicates whether the pair of entries in directories 0 and 1 contains the LRU, or the pair of entries in directories 2 and 3. Bit 1 indicates which of the pair consisting of entries 0 and 1 is the LRU, while bit 2 indicates the LRU within the pair consisting of entries 3 and 4. If any of the entries in the set of four is currently invalid, the new line (and its tag, or page address) will be placed in the invalid entry. Intel did not define which empty entry would be used if there were more than one in the selected set. If all of the entries are currently in use, the processor consults the LRU bits to select the entry to overwrite. The new line's page number is then stored in the tag field in the selected directory entry and the line is stored in the same line of the respective cache way. The LRU bits are then updated to reflect the new ranking amongst the four entries (i.e., the new LRU). The bit that provides the ranking for the pair that was just updated would be flipped to its opposite state to indicate that, within the entry pair, the other entry is now the LRU. Bit 0 is flipped to its opposite state, indicating that the other pair is now the LRU pair.

Code Cache Castout

When a line that was in the code cache is overwritten with a newly read line from a different page of memory, this is referred to as a *castout* of a cache line. Since code cache lines are always the same as memory, the castout line can just be erased and needn't be cast back to the L2 and to memory. The line may still be in the L2 cache (if it also hasn't been cast out to make room for a new line) and will remain there until the L2's LRU algorithm determines that it must be cast out to make room for a new line being read from memory. Intel doesn't state whether, if the line is no longer in L2, the line in the code cache is cast back to the L2 rather than just being invalidated.

Code Cache Snoop Ports

The L1 code cache is triple-ported. It receives read requests from the prefetcher through one of its ports. In addition, it implements two snoop ports:

Chapter 7: The Processor Caches

- **External snoop port.** When the processor latches a transaction request issued by another initiator, the address is forwarded to the internal caches for a lookup if it is a memory request. Refer to Table 7-1 on page 143 for the results of the snoop. Note that the table assumes that the line is not in the L1 data cache.
- **Internal snoop port.** Anytime a memory data write access is performed, the memory address is submitted to all three caches. If it results in a hit on the code cache, the code line is invalidated (S->I). For a detailed explanation, refer to the section entitled "Self-Modifying Code and Self-Snooping" on page 173.

Table 7-1: Results of External Snoop Presented to Code Cache

Type of External Transaction by Other Initiator	Initial State of Line in Code Cache	Snoop Result and Final State of Line in Code Cache
Memory read or write	I	<ul style="list-style-type: none"> • Indicate snoop miss to initiator. • Line stays in I state.
Memory read	S	<ul style="list-style-type: none"> • Indicate snoop hit to initiator. • Line stays in S state.
Memory write	S	<ul style="list-style-type: none"> • Indicate snoop miss to initiator (because the line is invalidated as a result of the write). • Invalidate line (S->I) because processor cannot snarf data being written to memory by another initiator.
Memory read and invalidate	I or S	<ul style="list-style-type: none"> • If line in I state, indicate snoop miss and line stays in I state. • If line in S state, indicate snoop miss (because the line is invalidated as a result of the read and invalidate) and line transitions from S->I state.

L1 Data Cache

As stated earlier, the size and structure of the L1 data cache is processor implementation-specific. The initial versions of the processor implement an 8KB, 2-way, set-associative data cache (pictured in Figure 7-3 on page 145), but as pro-

Pentium Pro Processor System Architecture

cessor core speeds increase, it is likely that the cache sizes will also be increased because the faster core can process code and data faster. Each of the data cache's cache banks, or ways, are further divided into two banks (more on this later).

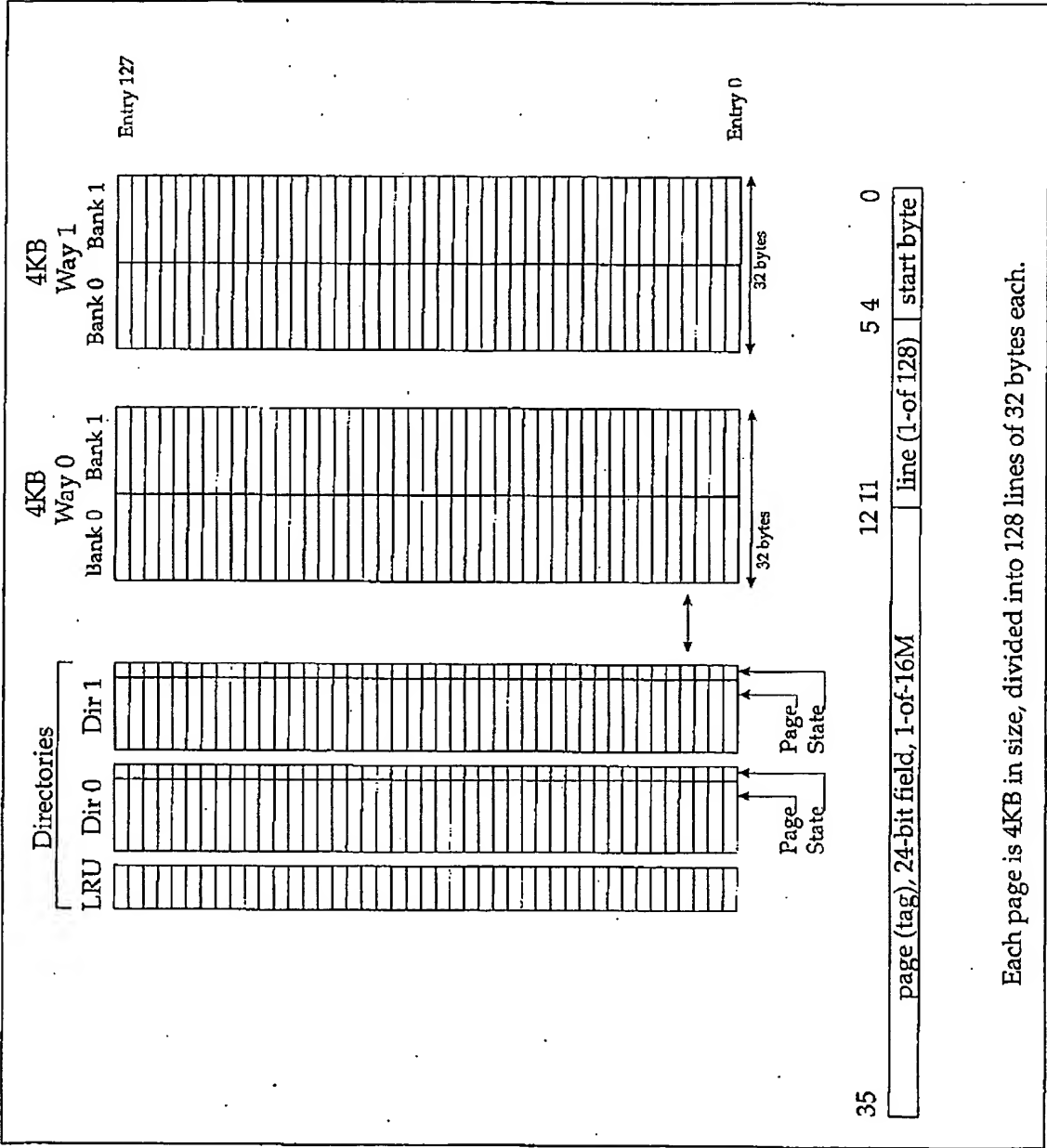
Data Cache Uses MESI Cache Protocol

The L1 data cache exists for only one reason: to service memory data read and write requests originated as a result of code execution. Each line storage location within the data cache can currently be in one of four possible states:

- **Invalid state (I).** There is no valid line in the entry.
- **Exclusive state (E).** The line in the entry is valid, is still the same as memory (i.e., it is fresh, or clean), and no other processor has a copy of the line in its caches.
- **Shared state (S).** The line in the entry is valid, still the same as memory, and one or more other processors may also have copies of the line (or may not because the processor cannot discriminate between reads by processors and reads performed by other, non-caching entities such as a host/PCI bridge).
- **Modified state (M).** The line in the entry is valid, has been updated by this processor since it was read into the cache, and no other processor has a copy of the line in its caches. The line in memory is stale.

Chapter 7: The Processor Caches

Figure 7-3: L1 Data Cache



Pentium Pro Processor System Architecture

Data Cache View of Memory Space

When performing a lookup, the data cache views memory as divided into pages equal to the size of one of its cache banks (or ways). Each of its cache ways is 4KB in size, so it views the 64GB of memory space as consisting of 16M pages, each 4KB in size. Furthermore, it views each memory page as having the same structure as one of its cache ways (i.e., a 4KB page of memory is subdivided into 128 lines, each of which is 32 bytes in size).

When a 36-bit physical memory address is submitted to the data cache by the processor core, it is viewed as illustrated in Figure 7-3 on page 145:

- The upper 24 bits identifies the target page (1-of-16M).
- The middle 7 bits identifies the line within the page (1-of-128).
- The lower 5 bits identify the exact start address of the target data within the line and is not necessary to perform the lookup in the cache.

Data TLB (DTLB)

The data TLB is incorporated in the front end of the data cache, translating the 32-bit linear address produced by the load and store units into the 36-bit physical memory address before the lookup is performed. The size and organization of the data TLB is processor design-specific. The current versions of the processor have the following geometry: the TLB for page table entries related to 4KB data pages is 4-way set-associative with 64 entries. for more information, refer to "Request for Cache and TLB Information" on page 364.

Data Cache Lookup

The target line number, contained in address bits [11:5], is used to index into the data cache directory and select a set of two entries to compare against. Each directory entry consists of a 24-bit tag field and a two-bit state field. The cache compares the target page number to each of the selected set of two entries that are currently in the valid state (E, S, or M).

Chapter 7: The Processor Caches

Data Cache Hit

If the target page number matches the tag field in one of the entries in the E, S, or M state, it is a cache hit. The data cache has a copy of the target line from the target page. The line is in the cache way associated with the directory entry, residing in the entry selected by address bits [11:5]. The action taken by the data cache depends on:

- whether the data access is a read or a write
- the current state of the line
- the rules of conduct defined for this area of memory.

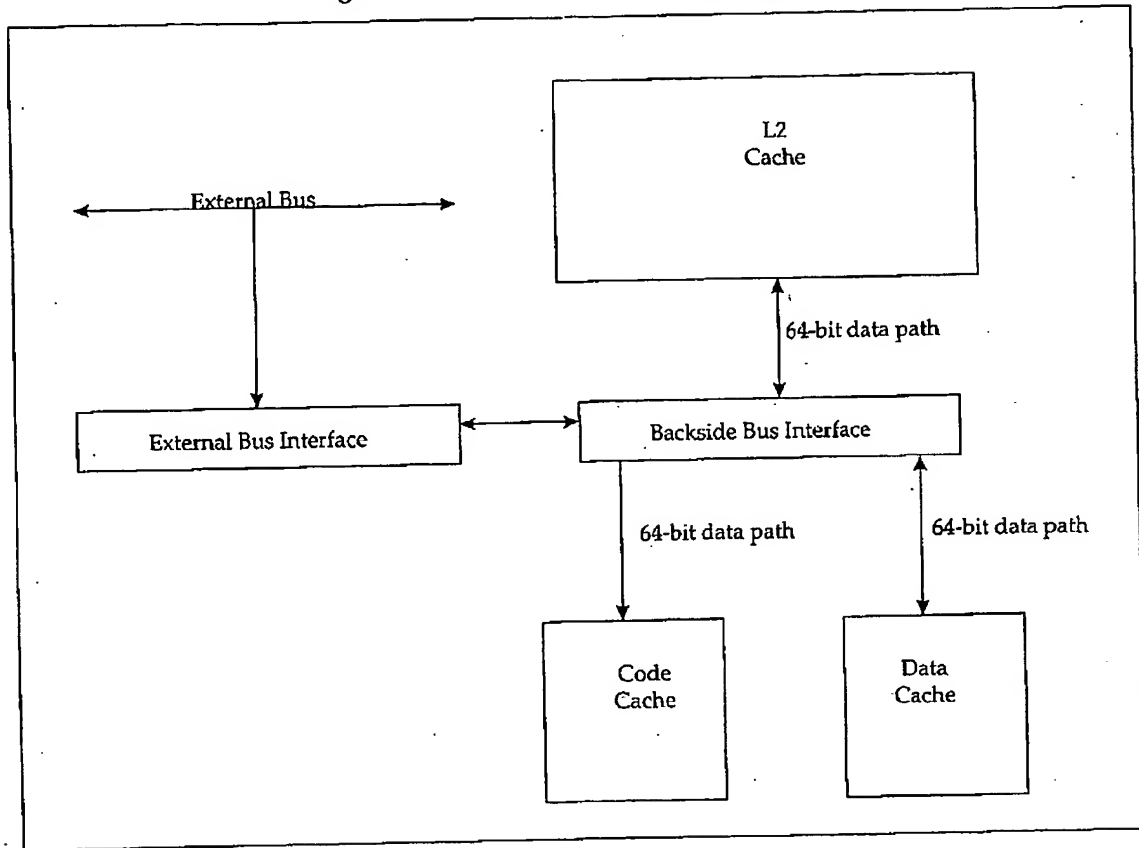
The sections that follow describe the actions taken by the data cache as a result of both cache hits and misses.

Relationship of L2 and L1 Caches

Refer to Figure 7-4 on page 148. The L2 cache acts as the backup for the L1 data and code caches. The sections that follow describe this relationship.

Pentium Pro Processor System Architecture

Figure 7-4: Relationship of L2 to L1 Caches



Relationship of L2 to L1 Code Cache

As stated earlier, the code cache only implements a subset of the MESI cache line states: S and I. The L2's relationship with the L1 code cache is therefore very simple. When the prefetcher request a line of code from the code cache, the request results in a hit or a miss.

In the event of a miss, the code cache issues a request to L2 cache. A lookup is performed in the L2 cache, resulting in a hit or a miss. If a hit, the requested line is supplied to the code cache one quadword at a time, critical quadword first. The line is placed in the code cache in the S state.

If the request results in an L2 cache miss, the L2 cache issues a request to the external bus interface and the line is read from external memory. During the transaction's snoop phase, caches in other processors report whether or not they have a copy of the line. If no other cache has a copy (snoop miss), the L2 cache

Chapter 7: The Processor Caches

places its copy in the E state upon completing the line read from memory. The code cache places its copy in the S state (the code cache only implements the S and I states). If any other cache has a copy in the E or S state, the line is read from memory and is placed in the L2 and code caches in the S state. If any other cache has a copy in the M state, the line is supplied from that processor's cache and is placed in the L2 and code caches in the S state. At the same time that the M line is supplied to this processor by the other processor, it is also written to memory. The line in the other processor's cache is therefore no longer different than memory. It transitions from the M to the S state.

Relationship of L2 and L1 Data Cache

Both the L2 and the L1 data caches are MESI caches. The relationship is therefore a little more complex than the L2's relationship with the code cache. The sections that follow provide a detailed description of the relationship. All of the cases described assume that the access is within a memory region defined as cacheable (i.e., the memory type is WT, WP, or WB). Furthermore, the discussion assumes that the L2 basically acts like a lookaside cache. As an example, assume that a line is read from memory and no one else has a copy (as indicated in the snoop phase of the transaction). If the area of memory is defined as WB, the line is placed in both caches (L2 and L1 data) in the E state, otherwise it's placed in the S state (lines from WT or WP memory are only stored in the S state). If any other processor indicates that it has a copy, the line is placed in both caches in the S state.

Intel does not document the interaction between the L2 and the data cache, however, so this discussion is based on hopefully intelligent speculation. It should be stressed that, even if this description is not the exact way that they have implemented it, it would work if designed this way. Furthermore, if they implemented it differently, it would be a variation on this discussion—not radically different.

Read Miss on L1 and L2

This discussion assumes:

- caching has been enabled (in CR0)
- the MTRRs have been initialized (defining some memory space as being cacheable)
- the read is within an area defined as cacheable (i.e., the memory type is WT, WP, or WB).

In the event of a miss on both caches, the L2 cache issues a request to the external bus interface to perform a 32 byte read from memory. When this read trans-

Pentium Pro Processor System Architecture

action is performed, the system's other caches report the state of their copies of the line during the transaction's snoop phase. There are three possibilities:

1. Miss on all other caches.
2. Hit on one or more caches on a clean copy of the line (S or E state).
3. Hit on one other cache on a modified copy of the line.

The sections that follow describe the actions taken in each of these cases.

Read Miss On All Other Caches. In this case, the line is read from memory. The state in which the line is stored depends on the rules of conduct defined within the memory area:

1. If defined as a WB area, the line is placed in the L2 and data caches in the E state.
2. If defined as a WT or WP area, the line is placed in the L2 and data caches in the S state.

Read Hit On E or S Line in One or More Other Caches. In this case, the line is read from memory. The line is placed in the L2 and data caches in the S state.

Read Hit On Modified Line in One Other Cache. In this case, the line is supplied by the cache that has the modified copy. At the same time that the line is being supplied to this processor's caches, it's accepted by the memory controller. Upon completion of the transfer, the line in the other processor's cache is therefore no longer different than memory, so the other processor transitions the state of its copy of the line from the M to the S state. In this processor's L2 and data caches, the line is placed in the S state.

Write Hit On L1 Data Cache

The following sections describe the actions taken when a data write is performed that hits on a line in the data cache. The actions taken depend on the state of the line and on the rules of conduct within the addressed memory region.

Write Hit on S Line in Data Cache. Unless, the area is designated as WP, the data cache line is updated. A write hit on an S line in a WP area has no effect on the data cache. Additional actions taken depend on the rules of conduct defined for the addressed memory area:

Chapter 7: The Processor Caches

1. In a WT or WP area, the line stays in the S state and the write data is posted in the processor's posted write buffer to be written back to memory later. If the line is in both the L2 and the code cache, the line in both caches is invalidated (see "Self-Modifying Code and Self-Snooping" on page 173). If the line is only in the L2 cache, it is invalidated.
2. In a WB area, the data cache line changes to the M state. The line was in the data and L2 caches in the S state, indicating that at least one other processor has a copy of the line. A read and invalidate transaction for 0 bytes is sent to the external bus to kill copies of the line in all other caches. If the line is in both the L2 and the code cache, the line in both caches is invalidated (see "Self-Modifying Code and Self-Snooping" on page 173). If it's only in the L2 cache, it remains in the S state, but is stale.

Write Hit On E Line in Data Cache. E copies of lines will only exist for regions of memory defined as WB. Lines from WT and WP areas are always stored in the S state. The write data is absorbed into the data cache line and the state of the line is changed from E to M. If the line is in both the L2 and the code cache, the line in both caches is invalidated (see "Self-Modifying Code and Self-Snooping" on page 173). If it's only in the L2 cache, it remains in the E state, but is stale.

Write Hit On M Line in Data Cache. M copies of lines will only exist for regions of memory defined as WB. Lines from WT and WP areas are always stored in the S state. The write data is absorbed into the data cache line and the line stays in the M state.

Write Miss On L1 Data Cache

The actions taken by the processor depend on the rules of conduct defined for the memory area:

1. If defined as a WT or WP area, the write is posted in the processor's posted write buffer to be written back to memory later. If the line is in L2 and it's a WT memory region, the L2 copy is updated. If it's a WP memory region, the L2 copy is unaffected.
2. If defined as a WB area, the processor uses an "allocate-on-write" policy. It issues a read request to the L2 cache. This results either in a hit or a miss on the L2 cache.
 - If it results in a hit on L2, the line is supplied to the data cache from L2. The line in L2 copy of the line is invalidated. The data cache copy is immediately updated and marked modified (i.e., I->M).
 - If it results in an L2 miss, the line is read from memory using a read and invalidate transaction to kill copies that may reside in other caches.

Pentium Pro Processor System Architecture

When the line is received, it isn't placed in the L2 cache, but is placed in the data cache, immediately updated, and placed in the M state (i.e., I->M).

L1 Data Cache Castout

When a data cache miss occurs, a read request is issued to the L2 cache. The line is then supplied to the data cache either from the L2 or from memory. In either case, the new line has to be stored in the data cache. If there aren't any empty entries in the selected set of cache entries, the data cache consults the LRU field (see "Data Cache LRU Algorithm: Make Room for the New Guy" on page 152) associated with the selected set of entries to determine which entry to castout to make room for the new line. The line being replaced (i.e., cast out), is in the E, S, or M state.:

- If the line is in the E or S state, it is still the same as the line in memory. When the line was read from memory, it was placed in both the L2 and the data caches. The line may or may not still be in the L2 cache (it may have been cast out to make room for a new line at some earlier point in time). If the line is still in L2, the line in the data cache is invalidated to make room for the new line. However, if the line is no longer in the L2 cache, the processor designers may have chosen to copy the line being cast out from the data cache to the L2 cache and invalidate it in the data cache, or may just invalidate it in the data cache. Intel doesn't define the action that is taken, but it's probably the former of the two.
- If the line is in the M state, it is copied from the data cache to the L2 cache (still in the M state) and is invalidated in the data cache to make room for the new line.

Data Cache LRU Algorithm: Make Room for the New Guy

When a miss occurs in the L1 data cache, the line is supplied either by the L2 cache or from external memory. In either case, the new line must be stored in the data cache. The discussion of the data cache lookup described how the target line number is used as the index into the cache directory and selects a set of two entries to compare against.

Assume that both of the selected set of entries are currently in use. In other words, the data cache has copies of the targeted line, but from two pages other than the desired page. A cache miss occurs. As stated earlier, the line will be supplied either from the L2 cache or from memory. When received, it should be obvious that the line must be stored in the data cache. It will be stored in the cache in the same relative position as it resided in within the memory page—in

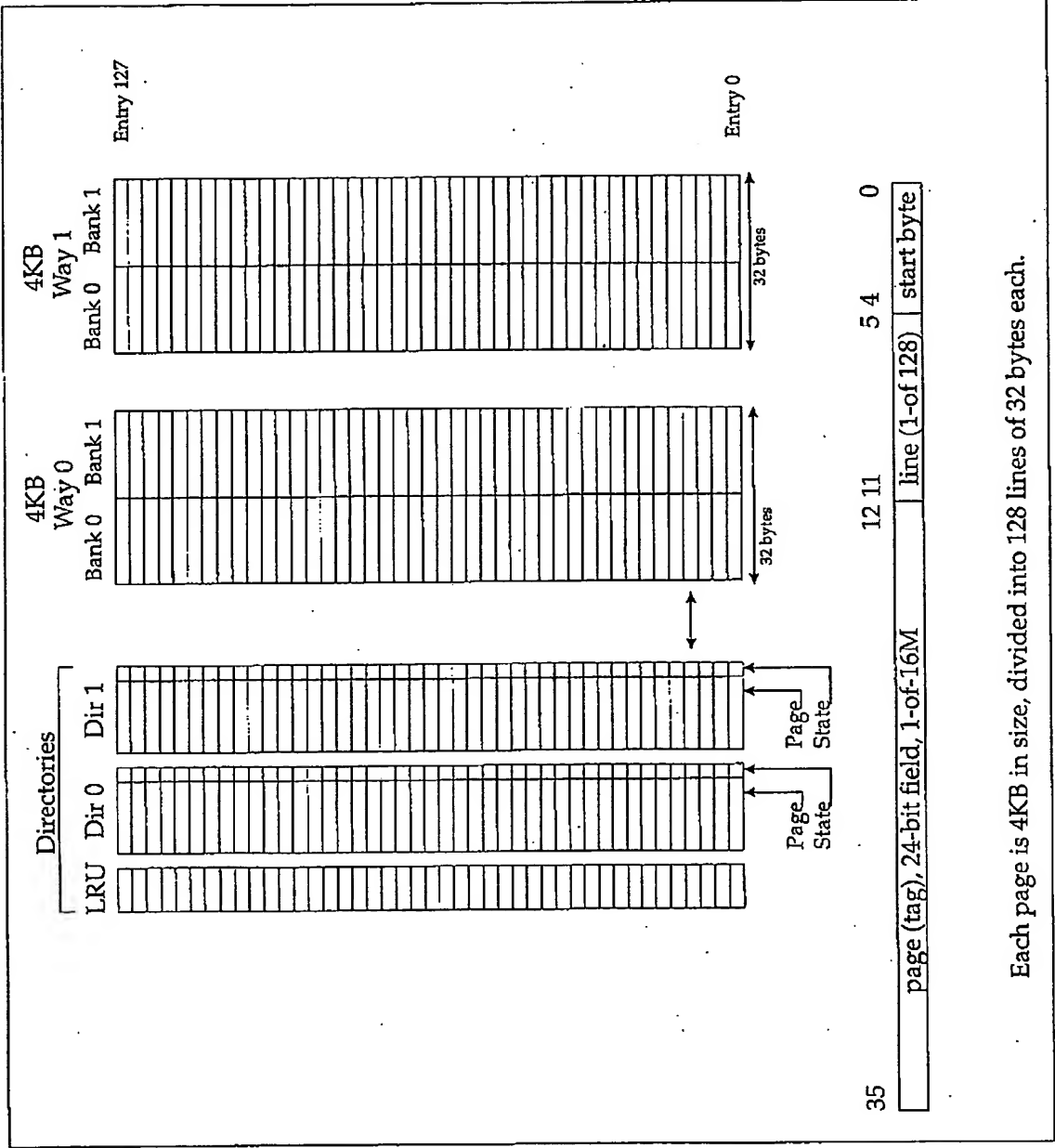
Chapter 7: The Processor Caches

other words, within the same line number in one of the two cache ways. One of the two entries in the selected set must therefore be overwritten (i.e., castout) with the new line and the new page number must be stored in the same entry of the associated directory.

Figure 7-3 on page 145 illustrates that each set of two directory entries has an associated LRU (Least-Recently Used) bit that is used to keep track of the least-recently used of the set of two entries. When both entries are currently valid (E, S, or M) and the same line of a different page is read in, the processor consults the LRU bit to determine which of the two entries to overwrite. The LRU bit selects the one to be overwritten. After the new entry is made, the LRU bit for the pair is flipped to its opposite state to reflect the new ranking (i.e., the entry that wasn't updated is now the LRU of the two). As described earlier, if the cast out line is in the M state, it is copied to the L2 cache.

Pentium Pro Processor System Architecture

Figure 7-5: L1 Data Cache



Chapter 7: The Processor Caches

Data Cache Pipeline

Figure 7-6 on page 156 illustrates the data and L2 cache pipeline stages.

1. When a load or a store is dispatched to the load or store execution units for execution, the address is submitted to the L1 data cache in the L1 stage.
2. During the next two clocks, the L2 and L3 stages, the lookup and, if a hit, the data transfer, takes place.
3. If the access is a miss in the data cache and its a load, it enters the first L2 cache pipeline stage, the L4 stage, where the address is submitted to the L2 cache for a lookup.
4. During the next two clocks, the L5 and L6 stages, the lookup and, if a hit, the data transfer of the critical quadword to the data cache, takes place.

It should be obvious that a new memory request can be submitted in each clock cycle. Refer to the example pictured in Figure 7-7 on page 156. In the example, all of the accesses are hits on the L1 data cache. For a discussion of the L2 cache, refer to the section entitled "Unified L2 Cache" on page 162.

1. **Clock one:** The address for access one is latched in the L1 stage.
2. **Clock two:** Access one advances to the L2 stage and access two's address is latched (L1 stage of access two). Access one's access to the cache begins.
3. **Clock three:** access one advances to the L3 stage and the cache is read or written. Access two advances to the L2 stage and its access to the cache begins. Access three enters the L1 stage and it's address is latched.
4. **Clock four:** access two advances to the L3 stage and the cache is read or written. Access three advances to the L2 stage and its access to the cache begins. Access four enters the L1 stage and it's address is latched.
5. **Clock five:** access three advances to the L3 stage and the cache is read or written. Access four advances to the L2 stage and its access to the cache begins. Access five enter the L1 stage and it's address is latched.

The cache is completing a data access request in each clock.

Pentium Pro Processor System Architecture

Figure 7-6: Cache Pipeline Stages

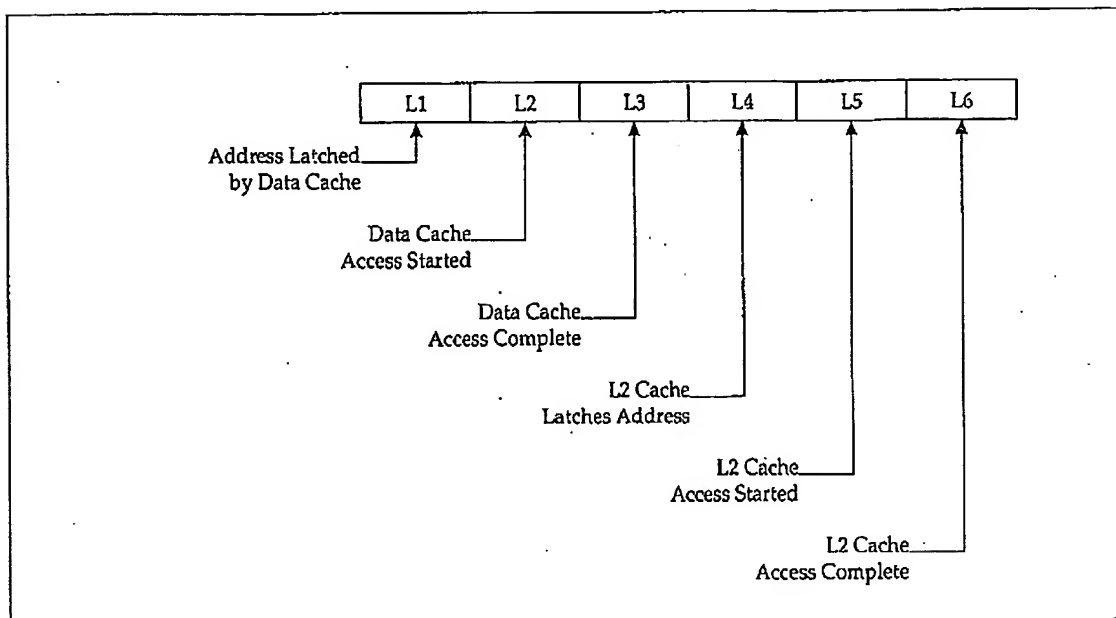
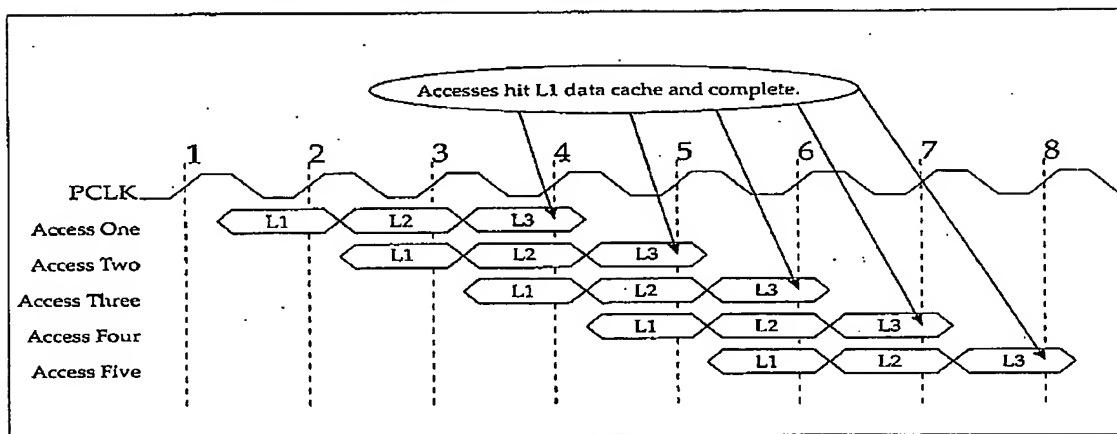


Figure 7-7: Example of Pipelined Data Cache Accesses



Data Cache is Non-Blocking

Earlier Processor Caches Blocked, but Who Cares?

In previous processors, a miss on the processor's cache resulted in a processor stall—no further memory requests could be submitted to the cache until the

Chapter 7: The Processor Caches

completion of the cache line fill from external memory. This means that data access requests generated by subsequent instructions that were executed could not be submitted for a lookup, thereby stalling the completion of those instructions. However, this was a moot point in the earlier Intel x86 processors because they always executed instructions in strict program order—if an instruction earlier in the program cannot complete due to a cache miss, the earlier processors would not proceed on to execute instructions that occur later in the program.

Pentium Pro Data Cache is Non-Blocking, and That's Important!

When a miss occurs on the Pentium Pro's data cache, however, the request is forwarded to the processor's L2 cache and the data cache is immediately available to service memory access requests (i.e., it does not block them) that might be generated by the execution of subsequent instructions in the program. This is extremely important in a processor that performs out-of-order (OOO) execution—it couldn't work without it! When the data associated with the data cache miss is subsequently returned from the L2 cache or memory, it is placed into the data cache, the data access request initiated by the earlier, stalled instruction completes, and the instruction becomes ready for retirement. There are some rules regarding the order in which the data cache will accept requests generated by out-of-order instruction execution:

- Loads generated by subsequent instructions can be serviced before a previously-stalled load completes.
- Loads generated by subsequent instructions can be serviced before a previously-stalled store completes, *as long as the load is from a different line.*
- Stores generated by subsequent instructions cannot be done before a previously-stalled store completes—*stores are always completed in order.*

Data Cache has Two Service Ports

Two Address and Two Data Buses

Refer to Figure 7-8 on page 158. The data cache has two, uni-directional data paths, each 64-bits wide, that connect it to the execution units:

- On a read that hits the data cache, the data cache output bus can deliver between one and eight bytes to the load execution unit per clock cycle.
- On a write that hits the data cache, the data cache input bus can deliver between one and eight bytes from the store execution unit to the data cache.

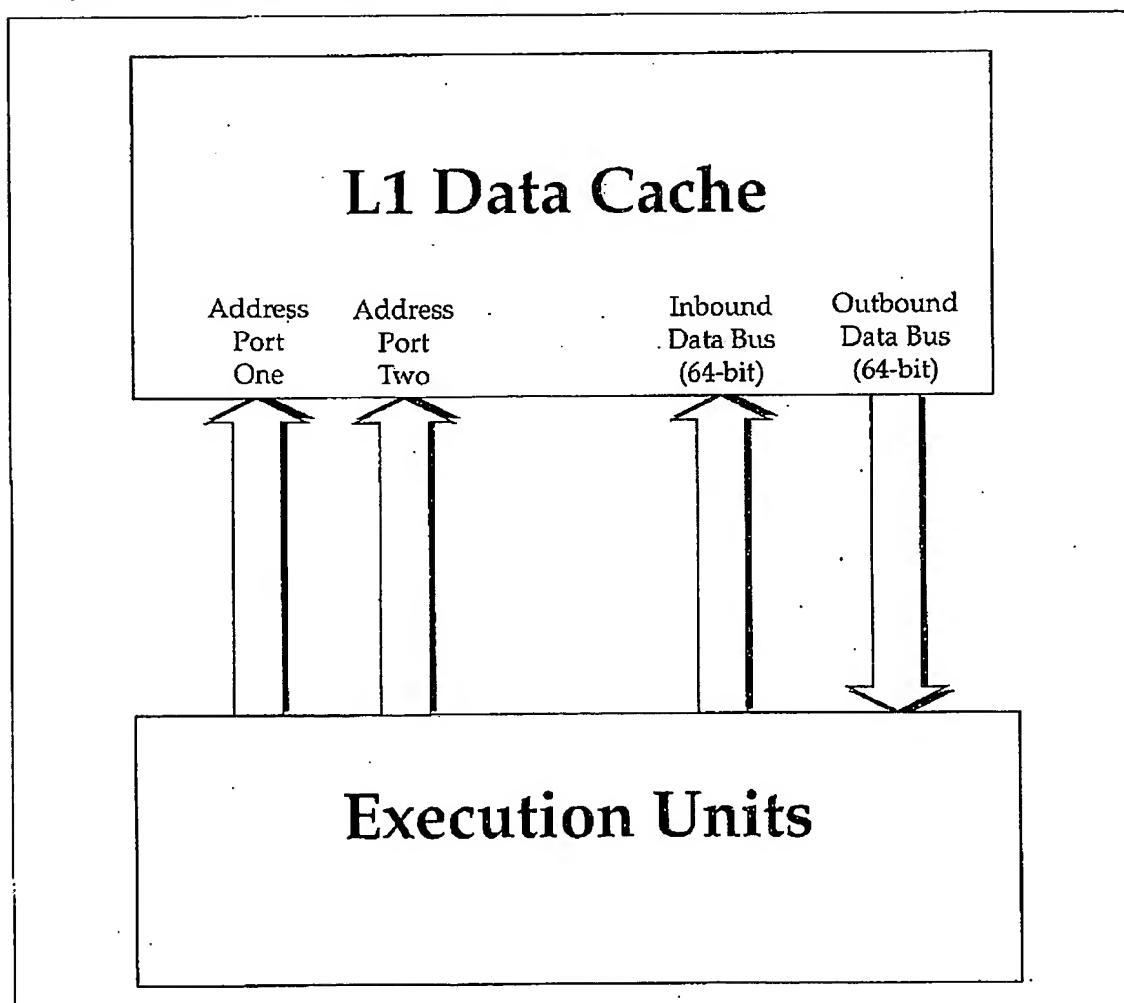
Pentium Pro Processor System Architecture

The cache also has two address buses so that it can simultaneously receive two addresses generated by two execution units (load unit and store address unit) for data cache lookups.

The result is as follows:

- The data cache can deliver the data for one load request per clock.
- The data cache can accept write data for one store request per clock.
- The data cache can simultaneously (i.e., in the same clock) deliver data for a load request and accept the data for a store request (but see "Simultaneous Load/Store Constraint" on page 159).

Figure 7-8: Data and Address Bus Interconnect between Data Cache and Execution Units



Chapter 7: The Processor Caches

Simultaneous Load/Store Constraint

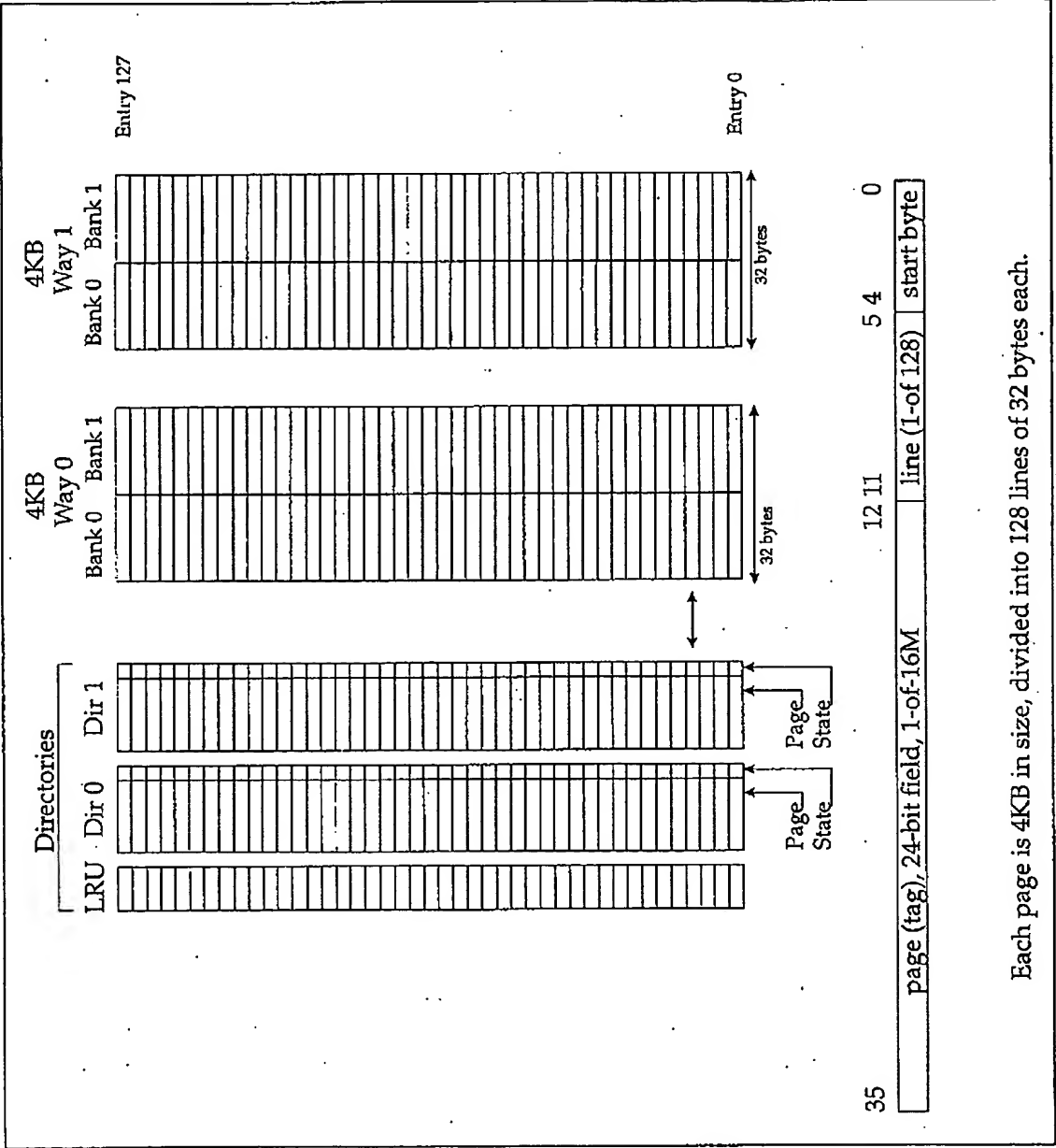
There is a constraint placed on the third scenario—the load and store can only be serviced simultaneously if they are in different cache ways or in the same cache way, but different line halves. Refer to Figure 7-9 on page 160.

Each of the data cache ways are divided into two, individually-addressable halves, or banks. If a load and store were simultaneously submitted to the cache, there are several possibilities:

- **The load and store hit in different cache ways.** The data cache can address the line in one way for the load and gate out the requested data onto its outbound data bus. It can simultaneously address the line in the other way and accept the write data arriving on the inbound data bus.
- **The load and store hit within the same way, but different halves.** The data cache can address the bank for the load and gate out the requested data onto its outbound data bus. It can simultaneously address the bank for the store and accept the write data arriving on the inbound data bus.
- **The load and store hit in the same half (i.e., bank) of the same way.** When the bank of SRAM is addressed, you can tell it that this is a read, or that it's a write, but you can't tell it that it's a read and write at the same time. The data pins on the SRAMs would have to simultaneously input and output data! It won't work. In this case, the data cache would service one request in one clock, and then the other request in the next clock.

Pentium Pro Processor System Architecture

Figure 7-9: Data Cache Structure



Chapter 7: The Processor Caches

Data Cache Snoop Ports

The L1 data cache is triple-ported. It can receive two simultaneous memory requests from the load and store execution units. In addition, it implements a snoop port. When the processor latches a memory transaction request issued by another initiator, the address is forwarded to the internal caches for a lookup if it is a memory request. Refer to Table 7-2 on page 161 for the results of the snoop.

Table 7-2: Results of External Snoop Presented to Data Cache

Type of External Transaction by Other Initiator	Initial State of Line in Data Cache	Snoop Result and Final State of Line in Data Cache
Memory read or write	I	<ul style="list-style-type: none">• Indicate snoop miss to initiator.• Line stays in I state.
Memory read	E	<ul style="list-style-type: none">• Indicate snoop hit to initiator.• Change state E->S.
Memory read	S	<ul style="list-style-type: none">• Indicate snoop hit to initiator.• No state change.
Memory read	M	<ul style="list-style-type: none">• Indicate snoop hit on modified line to initiator.• Make copy of line in writeback buffer and immediately write to memory (implicit writeback).• Change state of line from M->S.
Memory write	S or E	<ul style="list-style-type: none">• Indicate snoop miss to initiator (because the line is invalidated as a result of the write).• Invalidate line (S->I or E->I).
Memory write	M	<ul style="list-style-type: none">• Indicate snoop hit on modified line to initiator.• Unload line from cache to writeback buffer and invalidate line (M->I).• Immediately write line in writeback buffer to memory (implicit writeback).

Pentium Pro Processor System Architecture

Table 7-2: Results of External Snoop Presented to Data Cache (Continued)

Type of External Transaction by Other Initiator	Initial State of Line in Data Cache	Snoop Result and Final State of Line in Data Cache
Memory read and invalidate	S or E	<ul style="list-style-type: none">• Indicate snoop miss to initiator (because the line is invalidated as a result of the read and invalidate).• Invalidate line (S->I or E->I).
Memory read and invalidate	M	<ul style="list-style-type: none">• Indicate snoop hit on modified line to initiator.• Unload line from cache to writeback buffer and invalidate line (M->I).• Immediately write line in writeback buffer to memory (implicit writeback).

Unified L2 Cache

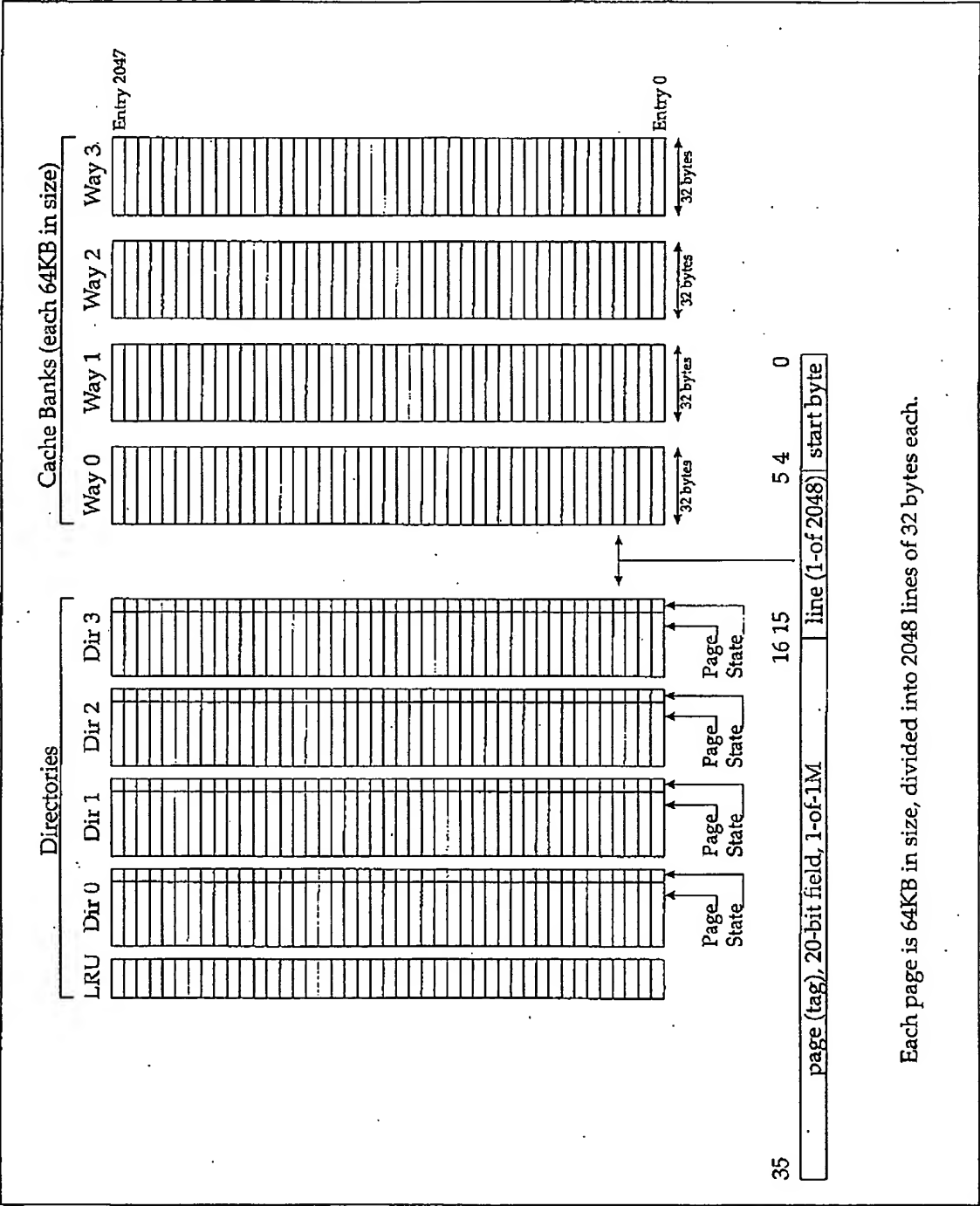
The current processor versions implement a 4-way set-associative unified L2 cache. The size and presence of the L2 cache is implementation-specific. The initial implementations are 256KB and 512KB in size.

L2 Cache Uses MESI Protocol

When the L1 code or data cache has a miss, it submits its request to the processor's L2 cache for a lookup. The L2 lookup results either in a hit or a miss. If it's a miss, the L2 cache issues a request to the external bus unit to fetch the line from memory. Meanwhile, the L2 cache continues to service requests issued by the L1 caches (like the data cache, it's non-blocking). Eventually, the line is fetched from external memory and is placed in the L2 and L1 caches. If the snoop result was a hit, the line is placed in the L2 cache in the S state. If the snoop resulted in a miss on all other caches, the line is placed in the L2 cache in the E state. Assuming that the data cache issued the request, the line is placed in the data cache in the E or the S state, depending on the snoop result. For more information on the relationship of the L2 and data caches, refer to "Relationship of L2 and L1 Caches" on page 147.

Chapter 7: The Processor Caches

Figure 7-10: 256KB L2 Cache



Pentium Pro Processor System Architecture

L2 Cache View of Memory Space

This explanation uses the 256KB L2 cache as the example. Note that the cache ways in the 512KB L2 would be twice as deep—4096 lines vs. 2048 lines—but is still 4-way set-associative.

When performing a lookup, the L2 cache views memory as divided into pages equal to the size of one of its cache banks (or ways). Each of its cache ways is 64KB in size, so it views the 64GB of memory space as consisting of 1M pages, each 64KB in size. Furthermore, it views each memory page as having the same structure as one of its cache ways (i.e., a 64KB page of memory is subdivided into 2048 lines, each of which is 32 bytes in size).

When a 36-bit physical memory address is submitted to the L2 cache by the code or data cache, it is viewed as illustrated in Figure 7-10 on page 163:

- The upper 20 bits identifies the target page (1-of-1M).
- The middle 11 bits identifies the line within the page (1-of-2048).
- The lower 5 bits identify the exact start address of the requested information within the line and is not necessary to perform the lookup in the L2 cache.

Request Received

An L1 cache (code or data) experienced a miss because one of the processor's units requested an item of information. The address that was submitted to the L1 cache identified the quadword needed (referred to as the critical quadword) and the required bytes within it (somewhere between one and eight bytes). Upon experiencing the L1 miss, the L1 cache forwarded the critical quadword address to the L2 for a lookup.

L2 Cache Lookup

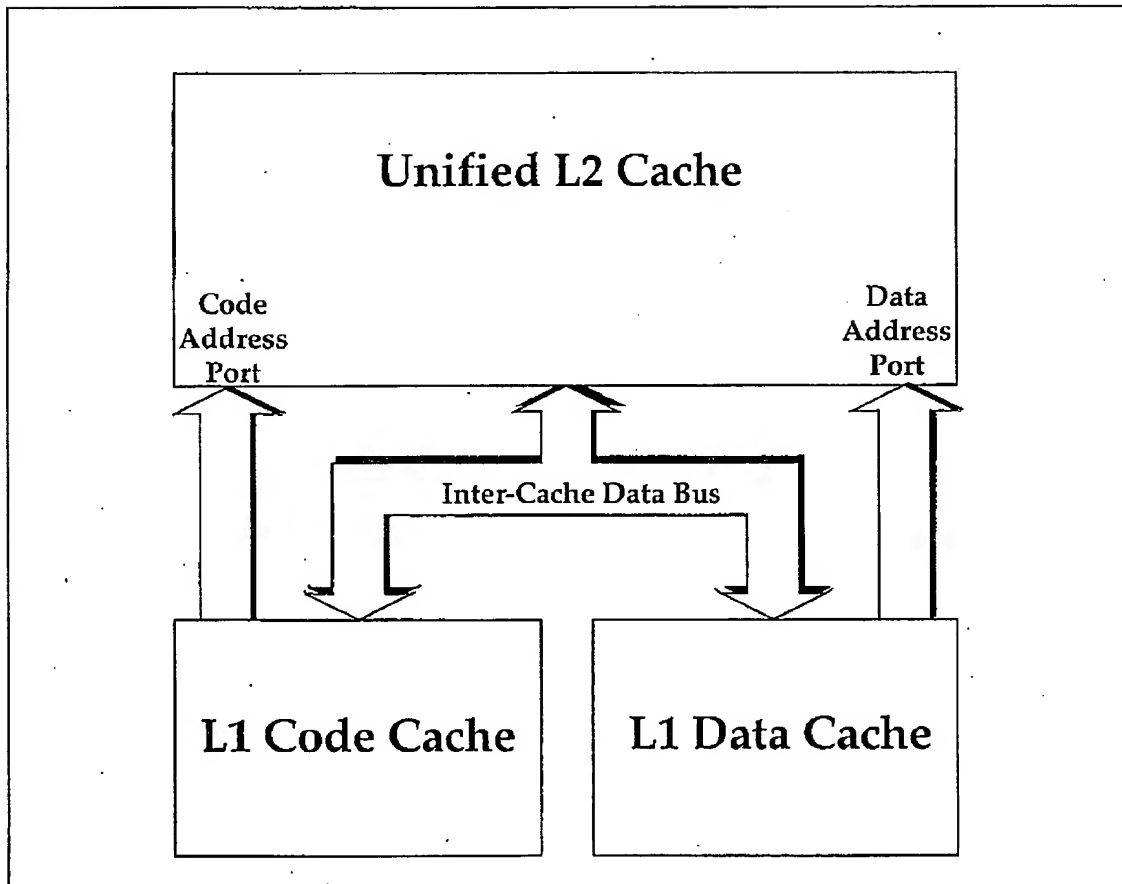
The target line number, contained in address bits [15:5] of the critical quadword address, is used to index into the L2 cache directory and select a set of four entries to compare against. Each directory entry consists of a 20-bit tag field and a two bit state field. The cache compares the target page number to each of the selected set of four entries that are currently in the valid state (E, S, or M).

Chapter 7: The Processor Caches

L2 Cache Hit

If the lookup results in a hit on the L2 cache, the requested line is sent back to the L1 cache that requested it. Because the data path between the L2 cache and the L1 caches is only 64-bits wide (see Figure 7-11 on page 165), the line is sent back to the L1 cache one quadword at a time. The critical quadword is always sent back first in order to satisfy and unstick the instruction that caused the miss. The other three quadwords that comprise the line are then sent back (Intel does not define the order in which they are transferred). As illustrated in Figure 7-11 on page 165, there is only data path to service both L1 caches. This means that only one of the L1 caches can be sent data at a time.

Figure 7-11: Data Path between L2 and L1 Caches



Pentium Pro Processor System Architecture

L2 Cache Miss

When an L2 cache miss occurs, the line must be fetched from memory. The backside bus unit issues a request to the external bus unit. The external bus unit arbitrates for the request bus and issues a request for the line, requesting the critical quadword first. The 32-byte line is supplied to the processor one quadword at a time, critical quadword first, in toggle mode transfer order (see "Toggle Mode Transfer Order" on page 171). The critical quadword is sent back to the requesting L1 cache immediately upon receipt (so that the execution unit that initiated the request can unstall). It is also placed in the entry selected in the L2 and L1 caches. The remaining three quadwords are transferred back to the processor and, when the entire line has been received, the L2 cache places it in the E or the S state, and the L1 cache places it in the appropriate state (if the code cache, it is placed in the S state; if the data cache, it is placed in the appropriate state—see "L1 Data Cache" on page 143).

L2 Cache LRU Algorithm: Make Room for the New Guy

When a cache miss occurs and a new line must be fetched from memory, it must then be stored somewhere in the L2 cache. When the cache lookup was performed, the target line number in bits [15:5] was used as the index into the cache directories, selecting a set of four directory entries to examine. If all four of the entries are currently valid but do not have a copy of the selected line from the targeted page, the cache uses the LRU bit field (see Figure 7-10 on page 163) associated with the selected set of four entries to determine which of the four to cast out to make room for the new line when it arrives from memory. The entry selected for castout is in the E, S, or M state.

If the castout line is in the E or S state, it can just be erased. If in the M state, however, it is the freshest copy of the line (the one in memory is stale). The processor unloads it from the cache (to make room for the new line) into a 32-byte writeback buffer and schedules it to be written back to memory. The writeback of the modified line is not considered to be a high-priority operation. If the processor has other writes that are considered more important, they will occur first.

Whenever another bus initiator starts a memory transaction, the processor must snoop the memory address in its caches. It should be obvious that it must also snoop it in its writeback buffers. If the initiator is addressing the modified line currently sitting in a writeback buffer waiting to be written back to memory, the

Chapter 7: The Processor Caches

processor will signal a hit on a modified line and immediately supply the data to memory from its writeback buffer. This is referred to as an implicit writeback of a modified line (see the chapter entitled "The Response and Data Phases" on page 277 for more information).

The L2 cache is a 4-way set-associative cache, as is the L1 code cache. As with the L1 code cache, Intel does not document how the LRU algorithm is implemented. For an explanation of the 486 method (which is also a 4-way set-associative cache), refer to "Code Cache LRU Algorithm: Make Room for the New Guy" on page 141.

L2 Cache Pipeline

Refer to Figure 7-12 on page 168. When a miss occurs in the L1 data or code caches, the critical quadword address is forwarded to the L2 cache for a lookup.

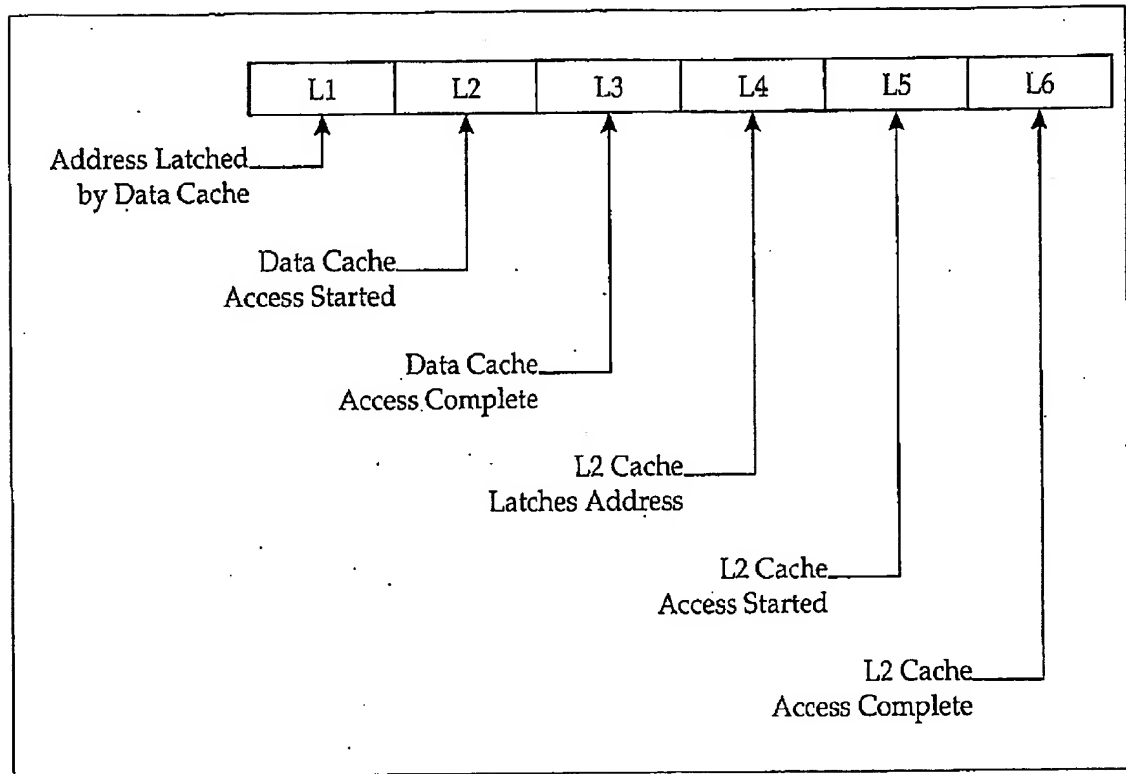
1. The address is latched by the L2 cache in the L4 stage.
2. The access to the L2 cache is started in the L5 stage.
3. The access to the L2 cache is completed in the L6 stage.

A load that hits the L1 data cache completes in 3 clocks, while one that misses L1 but hits the L2 completes in 6 clocks (if it results in a hit). If the load misses L2, it can't complete until the critical quadword arrives from memory.

It should be obvious that a new memory request can be submitted to the L2 cache in each clock cycle. Refer to the example pictured in Figure 7-13 on page 169. In the example, accesses one, three and four are data cache hits, while accesses two and five are misses on the L1 data cache and hits on the L2 cache. For a discussion of the L1 data cache, refer to the section entitled "L1 Data Cache" on page 143.

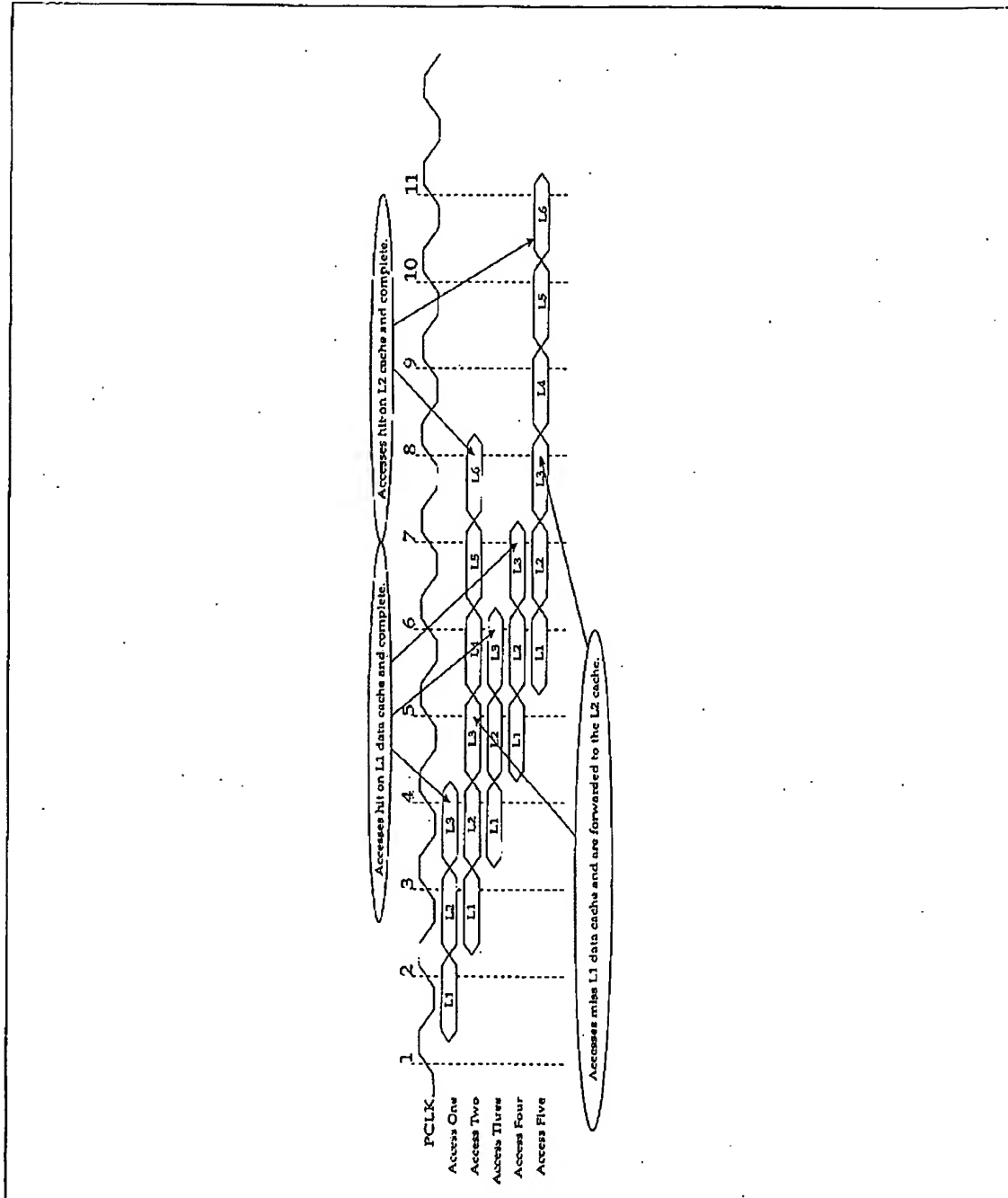
Pentium Pro Processor System Architecture

Figure 7-12: The L1 Data and L2 Cache Pipeline Stages



Chapter 7: The Processor Caches

Figure 7-13: Example of Pipelined L2 Cache Accesses Resulting in Hits



Pentium Pro Processor System Architecture

L2 Cache Snoop Ports

The L2 cache has two snoop ports:

- **Internal snoop port.** When a data write (i.e., a store) is performed that hits the L1 data cache, the address is submitted to the L2 cache via its internal snoop port and, if valid, the appropriate action is taken. Refer to "Relationship of L2 and L1 Caches" on page 147 and "Self-Modifying Code and Self-Snooping" on page 173.
- **External snoop port.** When the processor latches a transaction request issued by another initiator, the address is forwarded to the three internal caches for a lookup if it is a memory request. Refer to Table 7-3 on page 170 for the results of the snoop.

Table 7-3: Results of External Snoop Presented to L2 Cache

Type of External Transaction by Other Initiator	Initial State of Line in L2 Cache	Snoop Result and Final State of Line in L2 Cache
Any type	I	No effect on line state. Snoop result = miss.
Memory read	E or S	No effect on line state. Snoop result = hit.
	M	Snoop result = hit on modified line. Line is supplied to request agent from L2 cache and its state is changed from M->S.
Memory write	E or S	Line invalidated (S->I). Snoop result = miss.
	M	Snoop result = hit on modified line. Line is supplied to request agent from L2 cache and its state is changed from M->I.

Chapter 7: The Processor Caches

Table 7-3: Results of External Snoop Presented to L2 Cache (Continued)

Type of External Transaction by Other Initiator	Initial State of Line in L2 Cache	Snoop Result and Final State of Line in L2 Cache
Memory read and invalidate	E or S	Line invalidated (S->I). Snoop result = miss.
	M	Snoop result = hit on modified line. Line is supplied to request agent from L2 cache and its state is changed from M->I.

Toggle Mode Transfer Order

The processor transfers a full 32-byte cache line to or from memory under the following circumstances:

1. When a **miss** occurs on the **L1 and L2** caches, the line is read from memory.
2. When a **write miss** occurs on the data cache in a memory area designated as **WB**, the L1 data cache first attempts to read the line from the L2 cache. If it misses L2, the line is read from memory using the memory read and invalidate transaction.
3. When a new line is being read into the L2 cache from memory, the L2 cache's LRU algorithm is used to select which of the selected set of four entries it will be stored in. If the selected entry is currently occupied by a modified line from another area of memory, the L2 cache unloads (i.e., casts out) the modified line from the cache to make room for the new line. The modified line is placed in a writeback buffer and scheduled to be written back to memory. When the memory write is performed on the bus, it will write the full line to memory.
4. When the processor snoops an external transaction request initiated by another bus agent, it may result in a **snoop hit on a modified line** in the data cache. The processor supplies the data line to the bus agent (and changes the state of its line based on the type of transaction).

Whenever a full line is transferred over the bus, it is transferred in toggle mode order, critical quadword first. Refer to Table 7-4 on page 172. The transfer order of the four quadwords that comprise the line is based on the position of the critical quadword within the line. As an example, assume that an instruction must

Pentium Pro Processor System Architecture

read two bytes from memory starting at location 1012h. These two bytes reside in the quadword comprised of locations 1010h through 1017h. In turn, this quadword is the third quadword in the line that consists of locations 1000h through 101Fh—quadwords 1000h through 1007h, 1008h through 100Fh, 1010h through 1017h, and 1018h through 101Fh. The transfer order will be:

- First quadword transferred is the critical quadword that contains the two bytes requested by the execution unit—1010h through 1017h.
- Second quadword transferred is 1018h through 101Fh.
- Third quadword transferred is 1000h through 1007h.
- Fourth quadword transferred is 1008h through 100Fh.

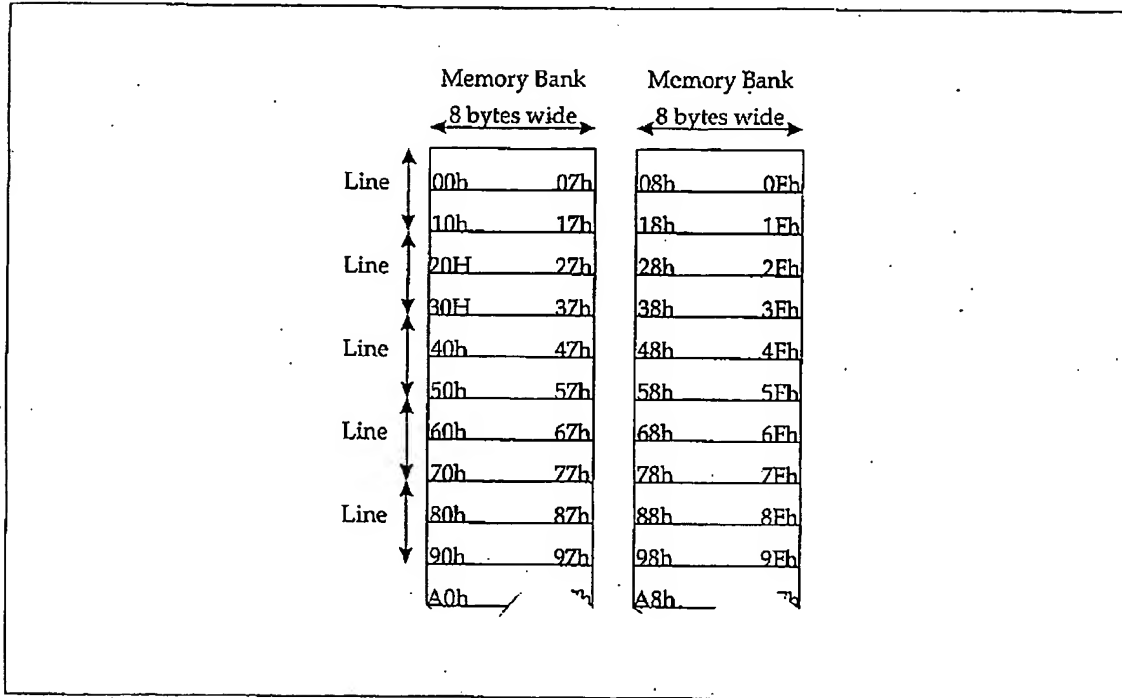
Table 7-4: Toggle Mode Quadword Transfer Order

In Critical Quadword	2nd quadword transferred	3rd quadword transferred is	4th quadword transferred is
If least-significant address 2 digits = 00h	one with least-significant 2 address digits = 08h	one with least-significant 2 address digits = 10h	one with least-significant 2 address digits = 18h
If least-significant 2 address digits = 08h	one with least-significant 2 address digits = 00h	one with least-significant 2 address digits = 18h	one with least-significant 2 address digits = 10h
If least-significant 2 address digits = 10h	one with least-significant 2 address digits = 18h	one with least-significant 2 address digits = 00h	one with least-significant 2 address digits = 08h
If least-significant 2 address digits = 18h	one with least-significant 2 address digits = 10h	one with least-significant 2 address digits = 08h	one with least-significant 2 address digits = 00h

The toggle mode transfer order is the one used by Intel since the advent of the 486 processor and is based on implementing main memory using an interleaved memory architecture (pictured in Figure 7-14 on page 173). Notice that the quadwords are woven, or interleaved, between the two banks. As each of the four accesses is performed, it is always the opposite DRAM memory bank that is being accessed. During each quadword access, the bank of memory that is not being accessed is charging back up after it was accessed. By the time the access to the current bank is completed, the other bank is all charged up and ready to accept another access. This bank-swapping results in good performance because the processor doesn't have to wait for the DRAM memory chips' pre-charge delay to elapse before accessing the next quadword in the series.

Chapter 7: The Processor Caches

Figure 7-14: 2-Way Interleaved Memory Architecture



Self-Modifying Code and Self-Snooping

Description

Normally the instructions that comprise a program are always executed as they were originally written by the programmer. Sometimes, however, the programmer chooses to start off executing an instruction or series of instructions as originally composed, but then later modifies the instruction(s) on the fly so that when executed the next time, it will execute in its modified form and have a different effect. This is accomplished by reading the instruction(s) into the processor's register set, modifying it, and then writing it back to memory. The next time the prefetcher fetches the modified instruction(s), they execute in their modified form and have the desired (i.e., altered) effect.

Generally speaking, self-modifying code is considered evil. Processor designers have to support it because people write it, but they usually don't optimize its performance—quite the opposite—you usually take quite a performance hit when it's executed. As one example, read the section entitled "The Internal

Pentium Pro Processor System Architecture

Snoop" in MindShare's book entitled *Pentium Processor System Architecture* (published by Addison-Wesley). For another, refer to the section on self-modifying code in MindShare's book entitled *PowerPC System Architecture* (published by Addison-Wesley).

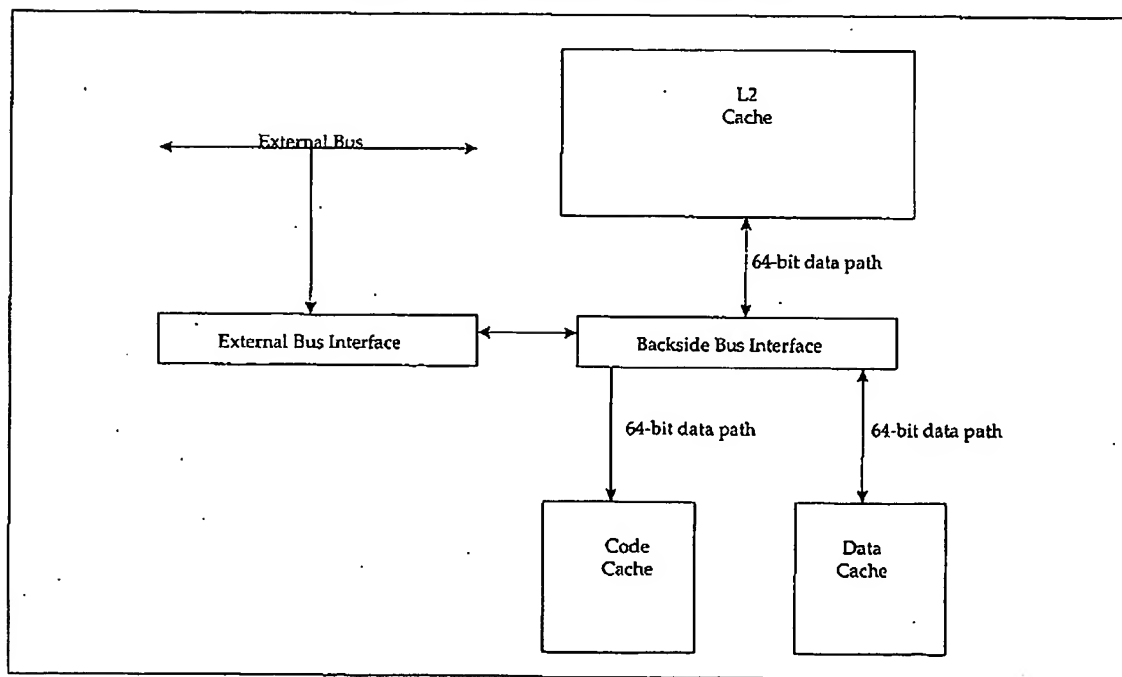
Refer to Figure 7-15 on page 175. The following numbered list defines the Pentium Pro's method for handling self-modifying code:

1. The first time that a piece of code is executed, the prefetcher issues a request for the line of code that contains the instructions.
2. If there is a code cache miss, the code cache issues the request to the L2 cache. If this also results in a miss, the L2 cache issues a request to the external bus unit to fetch the line from memory. When the line is returned from memory, it is placed in the L2 cache and is also placed in the code cache. The code cache, in turn, passes it on to the prefetch streaming buffer (what used to be called the prefetch queue).
3. The instructions are decoded into micro-ops and are executed in their original form.
4. When the programmer wishes to modify one or more instructions in this program, he or she uses a mov instruction to read the instruction into a register.
5. The memory read request is submitted to the data cache, resulting in a miss (the instruction is in the code cache, not the data cache).
6. The data cache issues a request to the L2 cache, resulting in an L2 cache hit. The L2 supplies the data (it's really code being temporarily treated as data) to the L1 data cache and the data cache supplies it to the load execution unit which places the data in the target register.
7. The programmer then modifies the instruction in the register.
8. Using a mov instruction, the programmer writes the data (i.e., the altered instruction) back to memory.
9. The write request is submitted to the L1 data, L1 code, and L2 caches as well as to the prefetch streaming buffer.
10. The code cache and L2 cache both have hits and invalidate their copy of the code line.
11. As the code line may have already been read into the prefetch streaming buffer again (due to a branch prediction), the processor also performs a lookup in the streaming buffer. If it's a hit, the buffer is flushed. Note that whereas the lookup in the caches is based on the physical memory address, the buffer lookup is based on the linear address (before the paging unit converts it to a physical address). This means that the programmer should always use the same linear address to fetch and to modify the instruction.
12. The data cache has a hit and updates its copy of the line. However, because it is also a hit on the code cache (the lookup is performed in all three caches simultaneously), the updated line is unloaded from the data cache (i.e., it's

Chapter 7: The Processor Caches

- placed in a 32-byte writeback buffer, invalidated in the data cache, and is scheduled to be written back to memory).
13. The next time that the prefetcher attempts to read the line (to execute the code again), it results in a miss on both the L1 code and L2 caches and the processor initiates a memory read transaction to read the line from memory.
 14. The processor snoops its own read transaction (*self-snooping!*) in its caches (it always does this, I just didn't mention it before) and in its writeback buffers. There are two possible cases: the modified line that was unloaded from the data cache has either already been written back to memory or it has not.
 - **Case A:** If the self-snoop doesn't result in a hit in its writeback buffers, the line has already been written back to memory and the processor reads the line from memory into its L2 and code caches.
 - **Case B:** If there is a hit on its writeback buffers, the line has not yet been written back to memory. The processor drives the modified line onto the bus during the data phase of the transaction and reads it back into the L2 and code caches *at the same time!* Main memory also receives the modified line.
 15. The code cache passes the line to the prefetch streaming buffer and the instructions are decoded and executed in their modified form.

Figure 7-15: Processor Cache Overview



Pentium Pro Processor System Architecture

Don't Let Your Data and Code Get Too Friendly!

Even though you may not write self-modifying code, the scenario described in the previous section can still occur, resulting in terrible performance for your program. Consider the case where you place some data in memory immediately following a program. The instructions at the end of your program and the data may reside within the same line. When you read any of these data items into a register, the entire line, comprised of mixed code and data, is read into the data cache. If you change the data in the register and then write it back to memory, it results in a hit on the data, code and L2 caches. The procedure described in the preceding section is then performed (in other words, the processor thinks that you are modifying code in the data cache).

ECC Error Handling

When a cache entry is made, ECC information is stored with the tag and with the line. Whenever the entry is read (during a lookup) the tag and the line are checked against their ECC data. If an error is incurred, the processor asserts its IERR# output. If the machine check exception is enabled (CR4[MCE] = 1), an error is reported in one of the machine check architecture register banks (see "Machine Check Architecture" on page 415). Alternately, the processor can be configured via its EBL_CR_POWERON MSR (see "Program-Accessible Startup Features" on page 45) to assert its BERR# output when an internal error occurs. When the chipset detects BERR# asserted, it can assert NMI to one of the processors to invoke the NMI handler. ECC information is also stored with the line of information and is checked for correctness whenever the line is read.

Hardware

Section 2:

Bus Intro and Arbitration

The Previous Section

The chapters that comprise Part 2, Section 1 focused on the processor's internal operation.

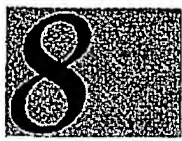
This Section

The chapters that comprise Part 2, Section 2 take a step outside the processor and introduce the processor's bus and transaction protocol. It consists of the following chapters:

- "Bus Electrical Characteristics" on page 179.
- "Bus Basics" on page 187.
- "Obtaining Bus Ownership" on page 201.

The Next Section

Part 2 Section 3 provides a detailed description of each phase that a transaction passes through from inception to completion.



Bus Electrical Characteristics

The Previous Chapter

The previous chapter provided a detailed description of the processor's L1 data and code caches, as well as its unified L2 cache. This included a discussion of self-modifying code and toggle mode transfer order during transfer of a cache line on the bus.

This Chapter

This chapter introduces the electrical characteristics of the Pentium Pro processor's bus.

The Next Chapter

The next chapter provides an introduction to the features of the bus and also introduces concepts critical to understanding its operation.

Introduction

One of the keys to a high-speed signaling environment is to utilize a low-voltage swing (LVS) to change the state of a signal from one state to the other. The Pentium Pro bus falls into this category. It permits the operation of the bus at speeds of 60MHz or higher. The bus is implemented using a modified version of the industry standard GTL (Gunning Transceiver Logic) specification, referred to by Intel as GTL+. The spec has been modified to provide larger noise margins and reduce ringing. This was accomplished by using a higher termination voltage and controlling the edge rates. The net result is that the bus supports more electrical loads (up to eight devices) than it would if implemented using the standard GTL spec. The sections that follow introduce the basic concepts behind the bus's operation. A detailed GTL+ spec can be found in the Intel processor data book.

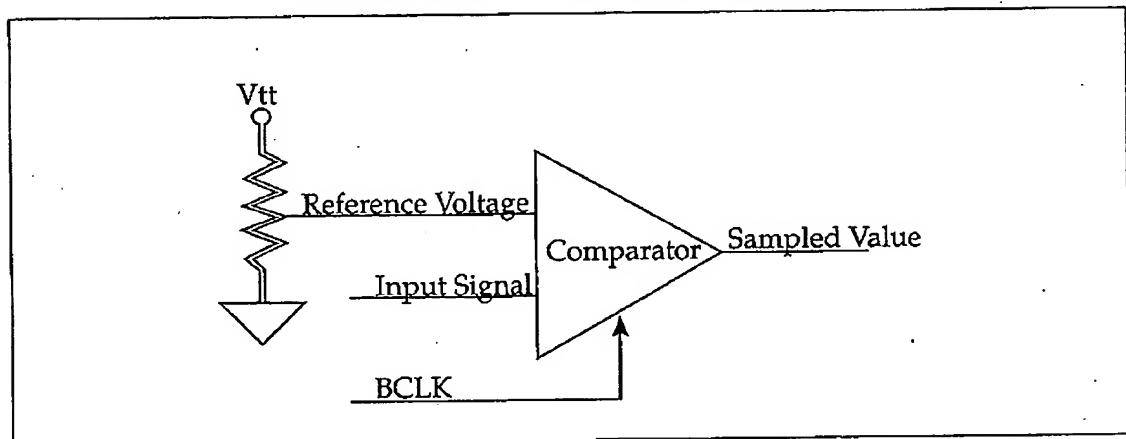
Pentium Pro Processor System Architecture

Everything's Relative

Every device that resides on the bus implements a comparator (i.e., a differential receiver) at each of its inputs. When a device samples the state of one of its bus inputs (only occurs on the rising-edge of BCLK), the value sampled (i.e., the voltage level) is compared to a standard reference voltage supplied to the device by a voltage divider network on the motherboard (see Figure 8-1 on page 180, where V_{tt} is the pullup, or termination, voltage). $V_{tt} = 1.5V_{dc} \pm 10\%$. If the sampled voltage is higher than the reference voltage by a sufficient margin, it represents an electrical high, while if it's sufficiently lower than the reference voltage, it represents an electrical low (the margins are pictured in Figure 8-2 on page 181 and are described in "How High is High and How Low is Low?" on page 184).

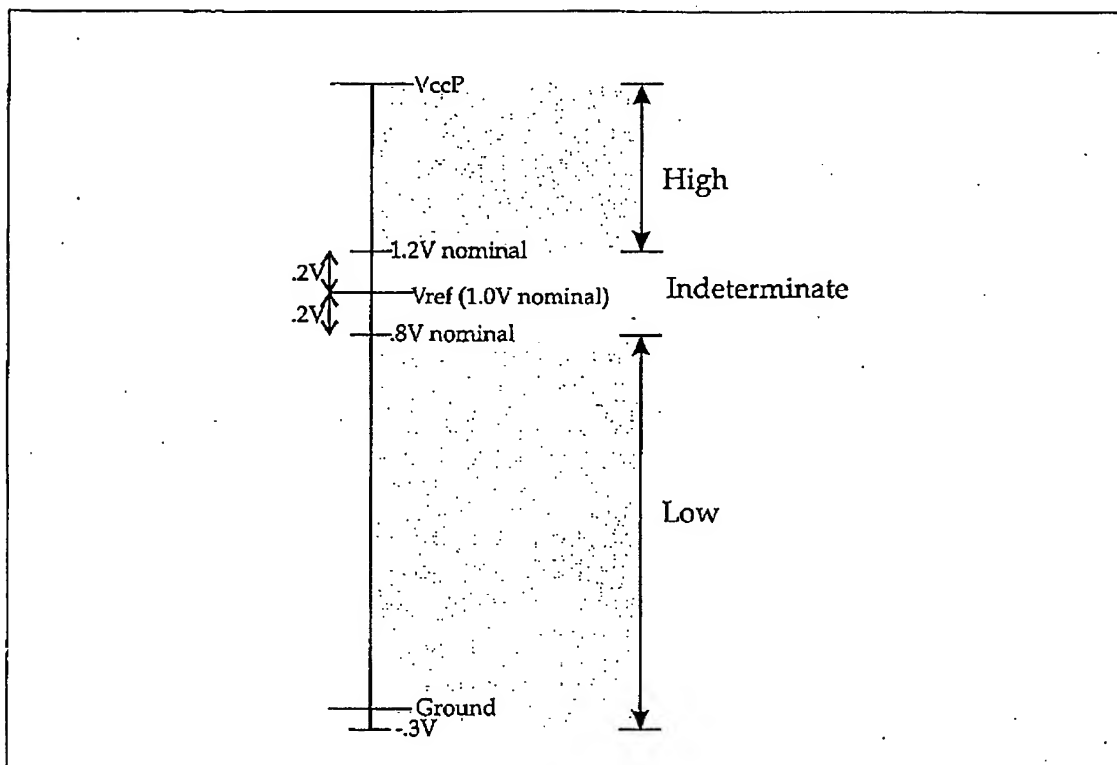
The great thing about this bus is that the differential receivers in bus devices determine highs and lows in a voltage-independent manner. The devices don't have to be redesigned if different voltage levels are used to indicate a high and a low— just supply a different reference voltage to the device!

Figure 8-1: Each GTL Input Uses a Comparator



Chapter 8: Bus Electrical Characteristics

Figure 8-2: Logic Levels



All Signals Active Low

All signals on the bus are open-drain, asserted low signals. This includes the data bus and the address bus! The address bus is designated as $A[35:3]\#$, the # sign indicating asserted when low. A logical one on an address signal is represented by an electrical low, and a logical zero as an electrical high. The same is true for the data bus, $D[63:0]\#$. *Just as a reminder (it was mentioned earlier in the book), Intel represents bus signal values in the data book's tables as logical values, not electrical values—so pay close attention to the fact that this is the reverse of their values when viewed on the bus.*

To place an electrical high on a signal, don't drive it. The pullup resistors (see Figure 8-3 on page 182) keep the line high. In other words, the only time that a device actually drives a GTL+ signal is when it asserts it to place an electrical low on the line. To deassert a signal, turn off the output driver. The pullups pull the line back high (very rapidly!).

Pentium Pro Processor System Architecture

Powerful Pullups Snap Lines High Fast

The values chosen for the termination pullups used on either end of a GTL+ trace (see Figure 8-3 on page 182) are such that they return the line back to the electrically-high state very rapidly. The signals therefore have very fast edge rates and tend to overshoot and ring for some time before settling. The resistor values for the pullups on the ends of a trace run match either:

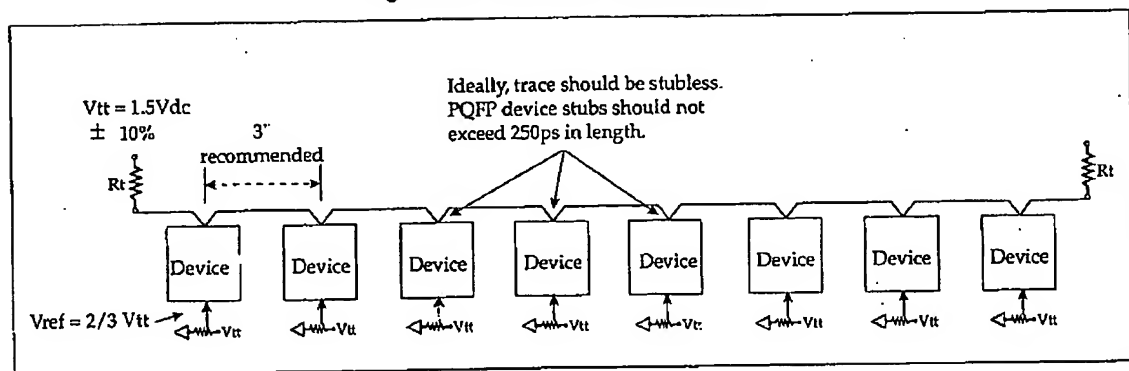
- the effective impedance of the trace run or
- the average impedance of all of the GTL+ trace runs.

The R_t (pullup resistor value) is typically in the range from 25 to 65 ohms.

The Layout

Refer to Figure 8-3 on page 182. Each signal line is daisy-chained between bus agents and is pulled up to V_{tt} on both ends. No stubs are permitted when connecting each device to the trace run. In reality, however, it is impossible to have a completely stubless connection to each of the bus agents. The only stub permitted is the connection to each device's pin pad and each of these should not exceed 250ps (the maximum allowable flight time for the signal to traverse the stub) in length. Maintaining 3" \pm 30% inter-device spacing minimizes the variation in noise margins between networks (note that this is a recommendation, not a rule).

Figure 8-3: GTL Bus Layout Basics



Chapter 8: Bus Electrical Characteristics

Synchronous Bus

A device starts to drive a signal on the bus on the rising-edge of BCLK and signals are only sampled on the rising-edge of the BCLK (Bus Clock) at the point where the BCLK rising-edge cross V_{tt} . BCLK is not a GTL+ signal. Rather, it is a 3.3V-tolerant signal and is supplied to all bus agents from a central source. This ensures that all of the bus devices are synchronized to the same clock.

Setup and Hold Specs

Refer to Figure 8-4 on page 184.

Setup Time

The minimum setup time is 2.2ns. Setup time is defined as: the minimum time from the input signal crossing the V_{ref} threshold (i.e., the reference voltage) on its way from a low to a high or a high to a low to when the BCLK signal's rising-edge crosses the V_{tt} threshold (i.e., the pullup value).

Hold Time

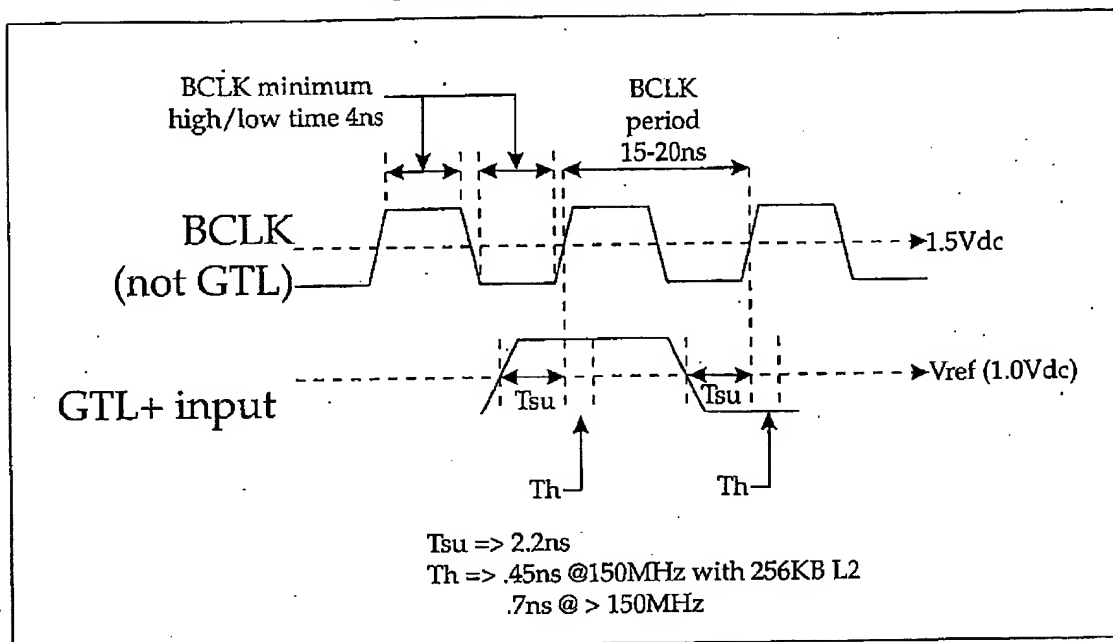
The minimum hold time is specified as:

- .45ns for the 150MHz processor with a 256KB L2 cache
- .7ns for other versions of the processor.

The hold time is defined as: the minimum time from BCLK crossing the V_{tt} threshold to when the input signal crosses the V_{ref} threshold.

Pentium Pro Processor System Architecture

Figure 8-4: Setup and Hold Specs



How High is High and How Low is Low?

When the value at an input is sampled (when the rising-edge of the BCLK crosses the V_{tt} level), it is determined to be a low or high as follows:

- $-3V_{dc} \leq \text{LOW} \leq V_{ref} - 2V_{dc}$
- $V_{ref} + 2V_{dc} \leq \text{HIGH} \leq V_{ccP}$

where $V_{ref} = 2/3$ of V_{tt} and V_{ccP} is the processor die's operating voltage. For the Pentium Pro processor, the V_{ccP} is specified as follows:

- For the 150Mhz processor with 256KB L2 cache: $2.945V_{dc} \leq V_{ccP} \leq 3.255$, typical = $3.1V_{dc}$.
- For all other processor versions: $3.135V_{dc} \leq V_{ccP} \leq 3.465V_{dc}$, typical = $3.3V_{dc}$.

$V_{tt} = 1.5V_{dc} \pm 10\%$.

$V_{ref} = 2/3$ of $V_{tt} \pm 2\%$.

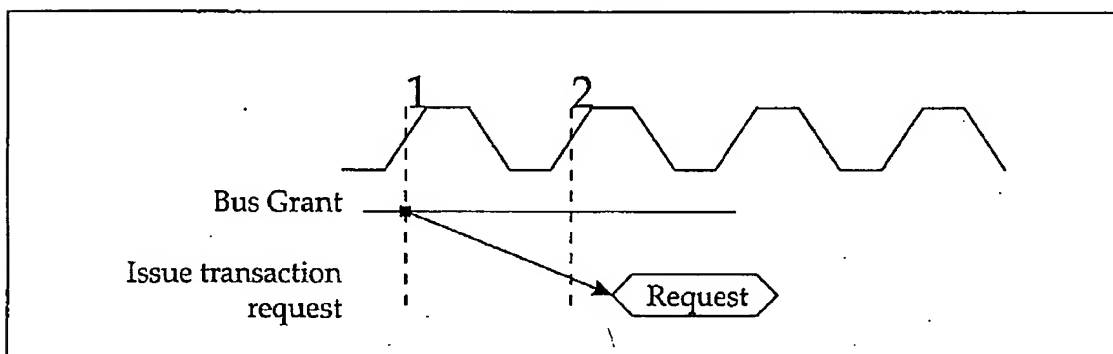
V_{ref} therefore = $1.0V_{dc}$.

Chapter 8: Bus Electrical Characteristics

After You See Something, You have One Clock to Do Something About It

The protocol is defined in a manner that always allows one clock after a condition is detected before an agent must present a reply. As an example, refer to Figure 8-5 on page 185. Assume that a processor detects that it has acquired bus ownership on the rising-edge of clock one. It takes action on this by assuming bus ownership starting on the rising-edge of clock two and driving out its transaction request onto the appropriate signals. It doesn't have to detect the condition on the rising-edge of clock one and present its reply in the same clock so that it will be seen by others on the rising-edge of clock two.

Figure 8-5: Example





Bus Basics

The Previous Chapter

The previous chapter introduced the electrical characteristics of the Pentium Pro processor's bus.

This Chapter

This chapter provides an introduction to the features of the bus and introduces terminology and concepts critical to an understanding of its operation.

The Next Chapter

Before a request agent can issue a new transaction to the bus, it must arbitrate for and win ownership of the request signal group. The next chapter covers all of the issues related to obtaining ownership of the request signal group. This includes:

- bus arbitration among symmetric agents (i.e., the processors).
- bus arbitration by the priority agent(s) and how this affects arbitration among the symmetric agents.
- How bus locking affects arbitration.
- How an agent or agents can block the issuance of new requests.

Agents

Agent Types

All devices that reside on the processor bus are referred to as *agents*. Basically, there are three types of agents:

- Request agent is the device that initiates a transaction (in other words, the initiator) by issuing a request (e.g., a memory read or write, or an IO read or write). It is commonly referred to as the transaction *initiator*.

Pentium Pro Processor System Architecture

- **Response agent** is the target of the transaction (e.g., an IO target or a memory target).
- **Snoop agents** are any devices on the bus that have memory caches (usually processors, but, as an example, in addition to the processors there could be an external, L3 cache that resides on the bus). Whenever any initiator starts a transaction, it is latched by all bus agents including the snoopers. If it is a memory transaction, the memory address is then submitted to the snoopers' caches for a lookup and the results of this "snoop" are reported back to the transaction initiator. The results will be one of the following:
 - **snoop miss**—doesn't have a copy of the addressed line at all.
 - **snoop hit**—one or more of the snoopers has a copy of the addressed line in the E or S state and it hasn't been changed since being read from memory.
 - **snoop hit on a modified line**—one of the snoopers has a copy of the line and one or more of the bytes in the line has been changed since the line was copied into the cache from memory. The line in memory is stale.

Multiple Personalities

Some agents are only capable of acting as response agents (i.e., the target of the transaction). As an example, the main memory controller typically acts as the target of memory reads and writes. It never initiates transactions, nor does it ever act as a snoop agent in a transaction.

Some agents are capable of acting as the response agent in some transactions and as the request agent for other transactions. As an example, in Figure 9-1 on page 190 one of the host/PCI bridges may:

- act as the response agent (i.e., the target) of a processor-initiated transaction to read data from an IO port in a PCI device beyond the bridge.
- act as the request initiator of a memory write transaction when a PCI master behind the bridge is writing data to main memory.

An agent may act as the request agent for transactions that it initiates and as the snoop agent for memory transactions initiators by others. An example would be a processor. It not only initiates transactions on an as-needed basis, but also snoops memory transactions that are initiated by the other processors or by the host/PCI bridges (for PCI masters).

Uniprocessor vs. Multiprocessor Bus

As noted in "Bus on Earlier Processors Inefficient for Multiprocessing" on page 26, the Pentium bus is ill-suited in a platform wherein multiple processors reside on the host bus (see Figure 9-1 on page 190) and must therefore compete for ownership of it when they need to perform a transaction.

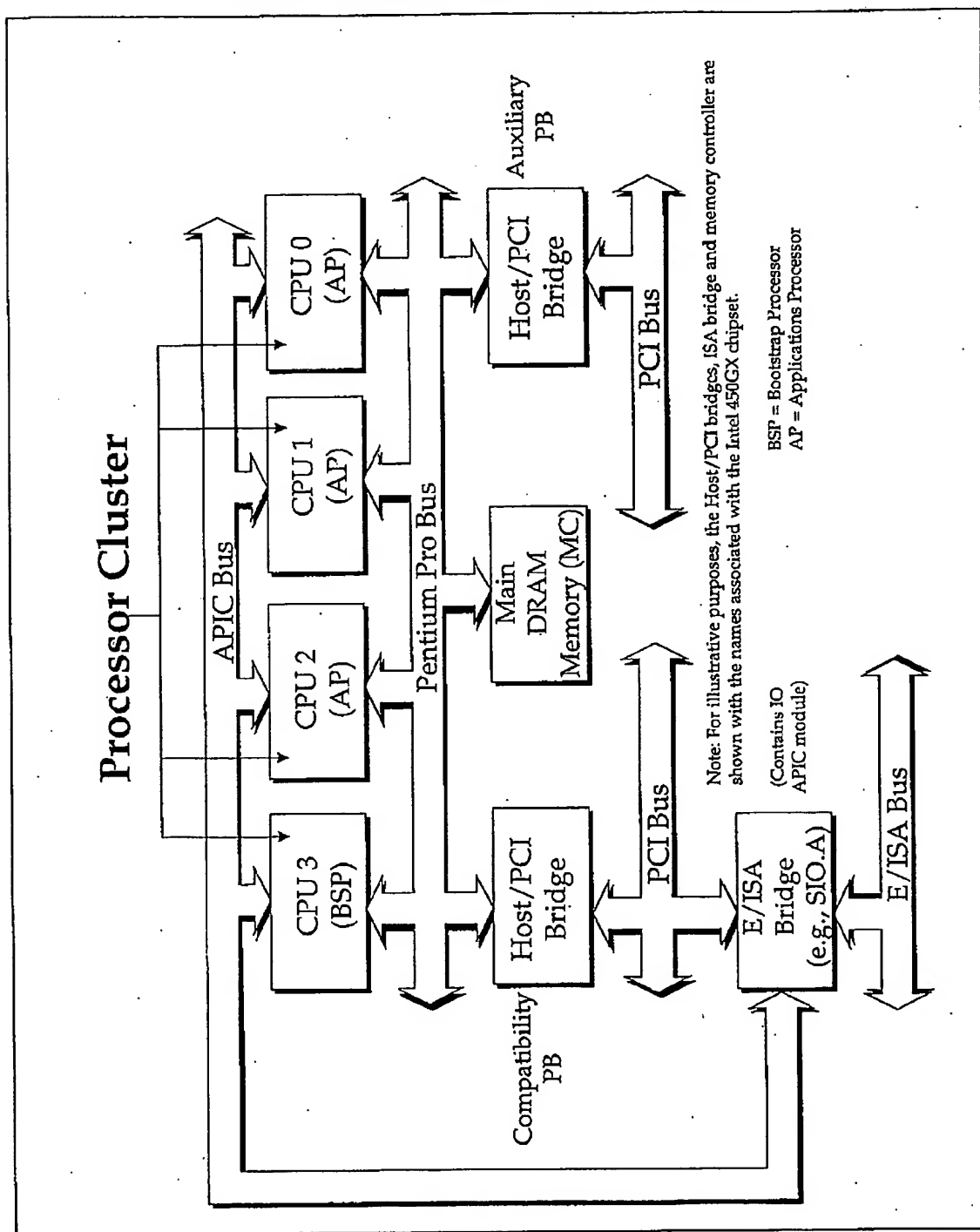
The Pentium Pro bus has been specifically designed to support multiple processors on the same bus. The following major changes have been made:

- In a typical Pentium Pro bus environment, up to eight transactions can be currently outstanding at various stages of completion.
- If the target of a transaction cannot deal with a new transaction right now (e.g., due to a logic busy condition), rather than tie up the bus by inserting wait states, it will issue a *retry* to the initiator. This causes the initiator to wait a little while and try the transaction again at a later time. This frees up the bus for other initiators.
- If the target of a read or write transaction realizes that it will take a fairly long time to complete the data transfer (i.e., provide read data or accept write data), it can instruct the initiator to break the connection and the target will later initiate a transaction to complete the transfer. This is referred to as *transaction deferral*.

These mechanisms prevent any properly-designed bus agent from tying up the bus for extended periods of time. A detailed description of the processor's bus is presented later in the book.

Pentium Pro Processor System Architecture

Figure 9-1: Block Diagram of a Typical Server System



Request Agents

Request Agent Types

There are two type of request agents:

- **symmetric request agents**—most typically, these are the processors. With regard to bus arbitration, the symmetric request agents have equal importance with respect to each other and use a rotational priority scheme for bus arbitration. Note that a custom-designed request agent other than a processor could be designed to operate as a symmetric agent. The symmetric agent bus arbitration scheme supports up to but no more than four symmetric request agents in the rotation.
- **priority request agents**—the system designer may include one or more request agents that are considered more important than the symmetric request agents. If a priority agent is competing against the symmetric agents for bus ownership, it wins and they lose.

Agent ID

What Agent ID Used For

When a request agent issues a transaction request, two of the items of information that it provides to the addressed response agent are:

- its unique agent ID
- a unique transaction ID

This information is only used by the response agent if it chooses to memorize the transaction and break the connection with the request agent. It then processes the read or write request off-line and, when the data has been read or written, it arbitrates for the bus and issues a *deferred reply transaction*. Using the agent and transaction IDs delivered earlier, it addresses the request agent that initially issued the read or write request and completes the transaction. A detailed description of deferred transactions can be found in "Transaction Deferral" on page 307.

Pentium Pro Processor System Architecture

How Agent ID Assigned

Each agent that is capable of initiating transaction requests must be assigned a unique agent ID at startup time. The agent typically accomplishes this by sampling one or more of its inputs on the trailing-edge of reset or when the power supply output voltages stabilize (i.e., on the rising-edge of the POWERGOOD signal). The manner in which the processors obtain their agent IDs is covered in "Agent ID Assignment" on page 204.

Transaction Phases

Pentium Transaction Phases

Each transaction initiated on the Pentium bus passes through three phases (or stages) from its inception to its completion:

- **Arbitration phase**—If the initiator doesn't currently own the bus, it requests ownership from the arbiter and awaits the granting of ownership.
- **Address phase**—After acquiring bus ownership, the initiator drives out an address and transaction type and asserts ADS# for one clock to indicate that a transaction has been initiated and a valid address and transaction type are on the bus. All agents latch the information and begin the decode to determine which of them is the target of the transaction.
- **Data phase**—The initiator waits for the target to provide the requested read data or to accept the write data.

Pentium Pro Transaction Phases

Each transaction initiated on the Pentium Pro bus proceeds through the following phases from its inception to its completion (note that some transaction types do not require a data transfer and therefore do not include the data phase):

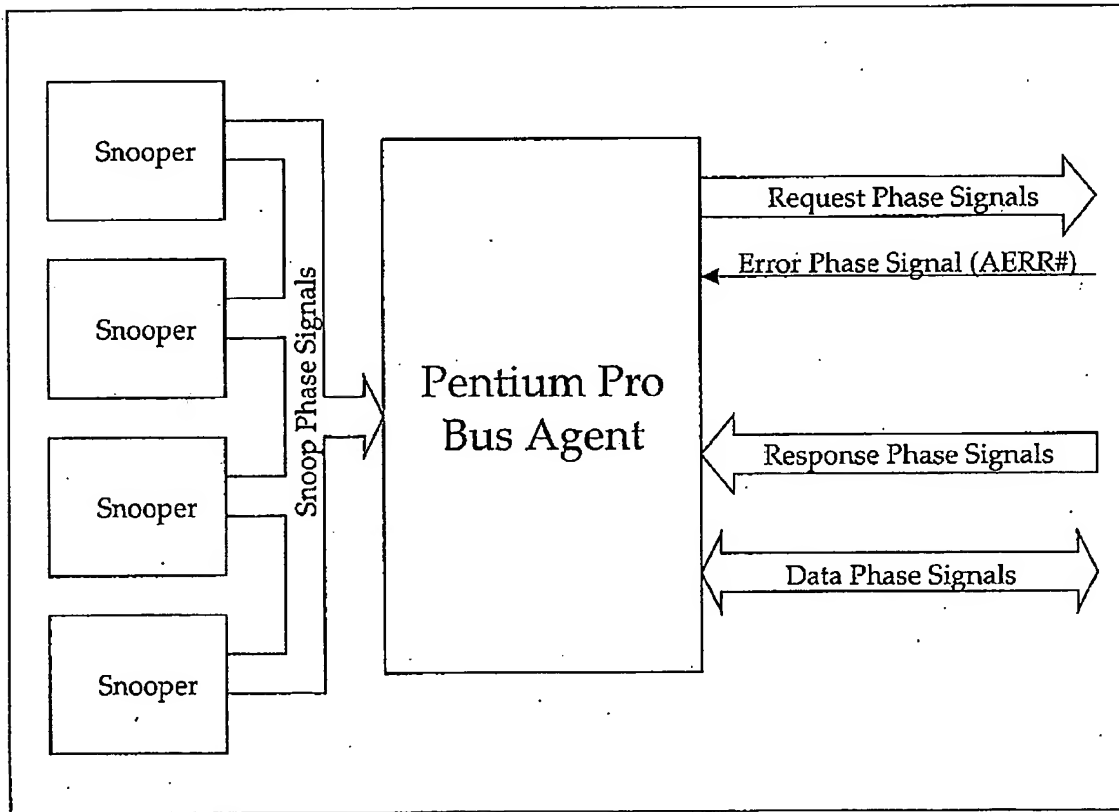
- Arbitration phase
- Request phase
- Error phase
- Snoop phase
- Response phase
- Data phase

Transaction Pipelining

Bus Divided into Signal Groups

Refer to Figure 9-2 on page 193. The bus is divided into signal groups. Each signal group is only used during a particular phase of the transaction.

Figure 9-2: Bus Signal Groups



Step One: Gain Ownership of Request Signal Group

The bus is divided into groups of signals, each of which is only used during a particular phase of a transaction. When a request agent wishes to acquire bus ownership in order to issue a transaction request, it arbitrates for ownership of the portion of the bus referred to as the *request signal group*.

Pentium Pro Processor System Architecture

Step Two: Issue Transaction Request

The transaction request is driven onto the request signal group during the request phase of the transaction. All of the other bus agents are required to latch the new transaction request. Once the request agent has completed delivery of the transaction request to the other bus agents, it ceases to drive the signals that comprise the request signal group. One BCLK period after the current request bus owner begins to turn off its drivers, another agent could take ownership of the request signal group and begin to drive out another transaction request.

Step Three: Yield Request Signal Group, Proceed to Next Signal Group

The request agent doesn't have to request ownership of the signal group used during the phase that follows the request phase. As the request agent finishes each phase, it relinquishes control of the signal group used in that phase and automatically takes ownership (or observes, as the case may be) of the signal group used in the next phase.

Phases Proceed in Predefined Order

The phases are passed through in a predefined order:

1. **Request phase.** Detailed description can be found in the chapter entitled "The Request and Error Phases" on page 241.
2. **Error Phase.** Detailed description can be found in the chapter entitled "The Request and Error Phases" on page 241.
3. **Snoop Phase.** Detailed description can be found in the chapter entitled "The Snoop Phase" on page 257.
4. **Response Phase.** Detailed description can be found in the chapter entitled "The Response and Data Phases" on page 277.
5. **Data Phase.** Detailed description can be found in the chapter entitled "The Response and Data Phases" on page 277.

The sections that follow provide a brief description of the phases.

Request Phase

The request agent uses the request signal group to issue the transaction request to the other bus agents. All of the other bus agents latch the request. The response agents (i.e., the targets) begin the decode to determine which of them is the target of the transaction. The snoop agents determine if it's a memory transaction. If it is, they must perform a cache lookup and deliver the snoop result during the snoop phase of this transaction. A detailed description of the request phase can be found in "Request Phase" on page 242.

Error Phase

The other bus agents check the parity of the request just latched. If the parity is correct, no action is taken. If one or more of them detect a parity error (in other words, the request was corrupted in flight), they each assert the AERR# signal for one BCLK at the end of the error phase (AERR# is a one of the few bus signals that is a multiple-owner, open-drain signal). The request agent checks the AERR# signal at the end of the error phase to see if they all received the request without error. If there was an error, the remaining phases of the transaction are cancelled (the request agent may choose to wait a few clocks and reattempt the transaction from scratch). If the request was received by all agents without error, the request agent proceeds to the snoop phase of the transaction. A detailed description of the error phase can be found in "Error Phase" on page 253.

Snoop Phase

After checking the state of the AERR# signal (and assuming that there wasn't an error), the request agent proceeds to the snoop phase and begins sampling the snoop result signal group. The snoop agents are responsible for delivering the result of their cache snoop during this phase. In addition, the response agent can assert the DEFER# signal if it intends to issue a retry or a deferred response during the response phase. Once the snoop result has been received, the request agent stops sampling the snoop result signals and the snoop agents cease driving them. A detailed description of the snoop phase can be found in "The Snoop Phase" on page 257.

Response Phase

Having completed the snoop phase, the request agent proceeds to the response phase and begins sampling the response signal group. The response agent (i.e., the target addressed by the request) is responsible for delivering its response (i.e., how it intends to handle the request) to the request agent during this

Pentium Pro Processor System Architecture

phase. Once the response has been received, the request agent stops sampling the response signal group and the response agent ceases to drive the response. The response delivered:

- tells the request agent to **retry** the transaction later.
- indicates a **hard failure** (don't retry and no data transfer; typically causes a machine check exception).
- if a write, it **will accept** the data in the data phase.
- if a read, it **will supply** the data during the data phase.
- if a snoop indicated a hit on a modified line, the entire line will be transferred from the snoop to memory (and, if its a read, to the request agent as well)—referred to as an **implicit writeback** response.
- instructs the request agent to end the transaction with no data transferred. The response agent will obtain or deliver the data off-line and will later initiate a deferred reply transaction to indicate completion. This is referred to as the **deferred response**.

A detailed description of the response phase can be found in "The Response and Data Phases" on page 277.

Data Phase

Most transactions involve a data transfer, but some don't. If this transaction involves a data transfer, the request agent proceeds to the data phase of the transaction. If it's a write transaction, the request agent takes ownership of the data bus and delivers the data to the response agent during this phase. If it's a read transaction, the response agent takes ownership of the data bus and delivers the data to the request agent. Once the data phase completes, the transaction has completed and the request and response agents stop using the data bus. A detailed description of the data phase can be found in "The Response and Data Phases" on page 277.

Next Agent Can't Use Signal Group Until Current Agent Done With It

When a request agent has finished driving its transaction request onto the request signal group, another request agent can take ownership of the request signal group and issue another transaction—and additional requests can be issued by the same or different request agents as the previous agent completes delivery of its request. This is pictured in Figure 9-3 on page 198.

Chapter 9: Bus Basics

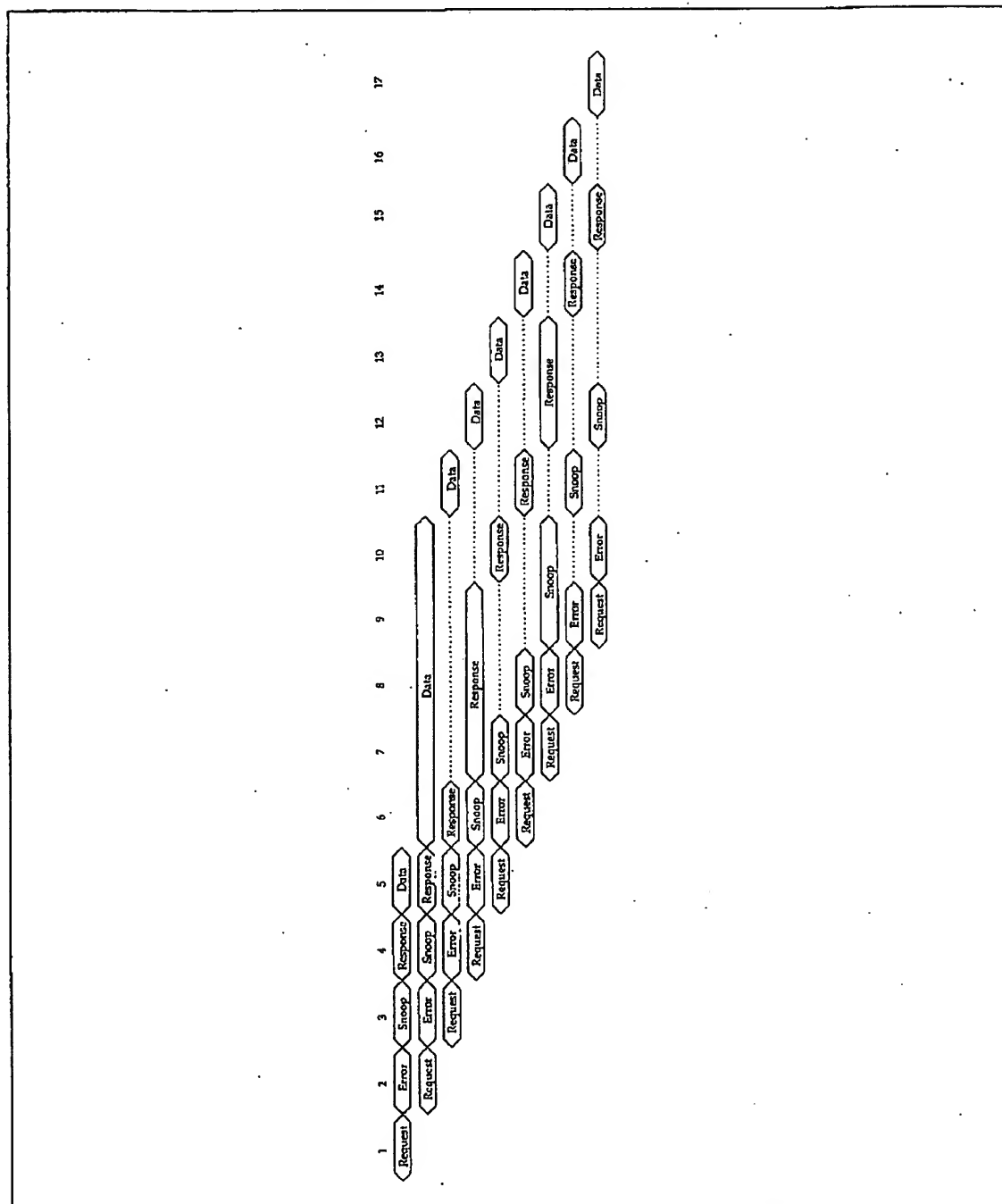
At a given instant in time, a number of transactions may be in progress at various stages of completion. As an example, at time 10 in Figure 9-3 on page 198 there are eight transactions active on the bus at various phases on their way to completion. The figure also highlights that some of the phases can take some time complete:

- the **snoop phase** (see times 9 and 10 in the figure) can be stretched by the snoopers if any of them needs a little time in order to produce the results of the cache lookup.
- the **response phase** (see times 8, 9 and 10) can be stretched by the currently-addressed target if it requires extra time before it decides what its response to the transaction will be.
- the **data phase** (see times 6 through 10) of the transaction cannot be completed until the request and response agents are ready to transfer the data.

In those cases where the current owner(s) of a signal group is not yet done using it, the next owner(s) cannot start using it until the previous agent's done and the signal group has been completely released.

Pentium Pro Processor System Architecture

Figure 9-3: Pipelined Transactions



Transaction Tracking

Request Agent Transaction Tracking

It should be obvious that a request agent must keep track of each transaction that it has issued as each passes through the various phases on its way to completion. The request agent has distinct responsibilities during each phase:

- issues transaction request during **request phase**.
- checks for good request transmission during **error phase**.
- waits for and checks snoop result during **snoop phase**.
- waits for and checks response from response agent during **response phase**.
- during the **data phase** of a read or write transaction, accepts or sources data.
- if in the response phase the response agent indicates that it wishes to defer completion until a later time, the request agent must keep track of this transaction until the response agent initiates a deferred reply transaction at a later time.

In order to fulfill these responsibilities, the request agent must keep track of all outstanding transactions issued by itself as well as those issued by other request agents. As each of its outstanding transactions enters a new phase, it must interact with the appropriate signal group.

Snoop Agent Transaction Tracking

The group of snoop agents that reside on the bus must also keep track of each transaction currently outstanding on the bus.

- At the appropriate times, they must present the snoop result for each transaction. As illustrated in Figure 9-3 on page 198 (in time 11), sometimes the delivery of the snoop result for a particular transaction must be delayed until all of the snoopers have presented the snoop results for the previous transaction.
- When a snoop results in a hit on a modified line, the snooper must be prepared to writeback the modified line to the currently-addressed memory response agent during the data phase of the transaction.

Pentium Pro Processor System Architecture

Response Agent Transaction Tracking

A response agent must track the following:

- it must latch each new transaction request and determine if it is the target of the transaction.
- if it is the target and it intends to issue a retry or a deferred response during the response phase, it must assert **DEFER#** during the snoop phase.
- if it is the target, the response agent must indicate how it wants to handle the transaction during the response phase of the transaction.
- if it is the target and this is a read, the response agent will have to return data to the request agent when the data phase is entered.
- if it is the target and this is a write, the response agent must be prepared to accept the write data when the data phase is entered.
- if it is the target and the snoop result is a hit on a modified line, the response agent must be prepared to accept the line from the snooper during the data phase of the transaction.

The IOQ

In order to properly interact with the bus at each stage of the appropriate transactions, each bus agent must maintain a record of all transactions currently in progress, what phase each is currently in, and what responsibilities (if any) it has during each phase. This record is kept in a buffer referred to as the agent's in-order queue, or IOQ. When each transaction receives a response guaranteeing that the transaction will be completed now, the transaction is deleted from the queue.

10

Obtaining Bus Ownership

The Previous Chapter

The previous chapter provided an introduction to the features of the bus and introduced terminology and concepts critical to an understanding of its operation.

This Chapter

Before a request agent can issue a new transaction to the bus, it must arbitrate for and win ownership of the request signal group. This chapter covers all of the issues related to obtaining ownership of the request signal group. This includes:

- bus arbitration among symmetric agents (i.e., the processors).
- bus arbitration by the priority agent(s) and how this affects arbitration among the symmetric agents.
- How bus locking affects arbitration.
- How an agent or agents can block the issuance of new requests.

The Next Chapter

This chapter described the acquisition of the request signal group. Once ownership has been acquired the request agent initiates the request phase of the transaction. The next chapter provides a detailed description of the request and error phases of any transaction.

Request Phase

There are a number of references to the request phase of the transaction in this chapter. After a request agent has arbitrated for and won ownership of the request signal group, it may then initiate a transaction by issuing a transaction

Pentium Pro Processor System Architecture

request during the request phase of the transaction. This consists of the output of two packets of information and the assertion of ADS# (Address Strobe) during the output of the first packet. For a detailed description of the request phase, refer to the chapter entitled "The Request and Error Phases" on page 241.

Symmetric Agent Arbitration—Democracy at Work

A symmetric system is one in which any processor is capable of handling (i.e., executing) any task. The job of the SMP (symmetrical multiprocessing) OS is to attempt to keep all of the processors busy at all times (in other words, executing various tasks). At a given instant in time, one or more of the processors may require ownership of the request signal group in order to communicate with an external device. In a well-designed system, the bus arbitration scheme used to decide which of the processors gets ownership next is based on rotational priority—each of the processors has equal importance.

No External Arbiter Required

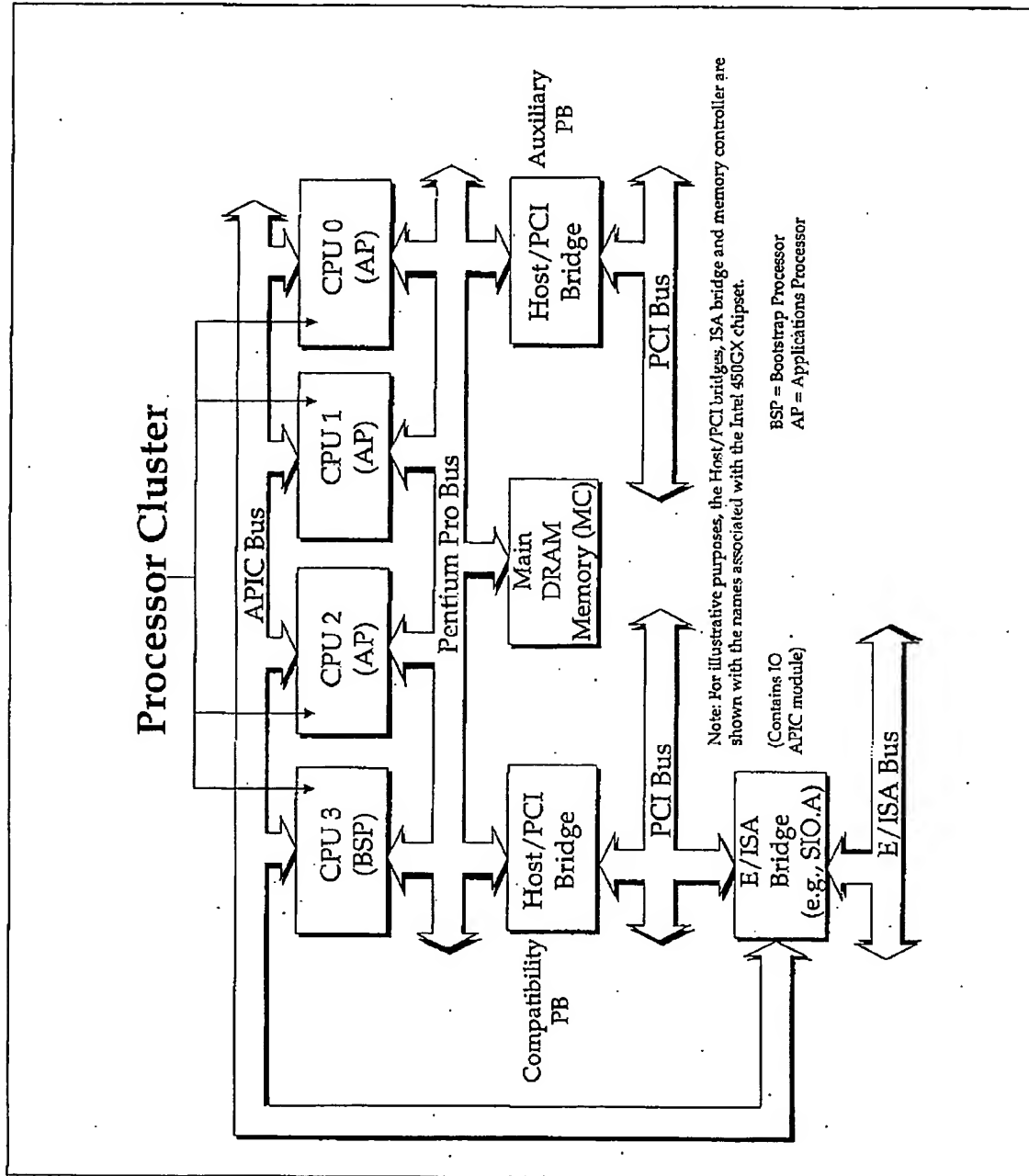
Refer to Figure 10-1 on page 203. The Pentium Pro processors that make up a cluster (i.e., the group of processors that reside on the processor bus) have a built-in rotational priority scheme. No external arbitration logic is necessary to decide which of them requires the request signal group and which gets it next. Each of the processors always keeps track of:

- whether any of them currently owns the signal group
- which of them owned the group last (or still owns it)
- and which of them gets to use it next (assuming it is asking for ownership).

In order for them to track this information, each must know its own agent ID as well as the agent ID of the processor that last gained ownership of the request signal group. If you know who had ownership last (or still has it), then you know the agent ID who's turn it is next (because it's a rotational scheme).

Chapter 10: Obtaining Bus Ownership

Figure 10-1: System Block Diagram



Pentium Pro Processor System Architecture

Agent ID Assignment

Each processor is assigned its agent ID automatically at the trailing-edge of reset when it samples the state of its BR[3:1]# inputs (see the detailed description in "Processor's Agent and APIC ID Assignment" on page 42).

Arbitration Algorithm

Rotating ID

As stated earlier, each processor must keep track of which of them gained request signal group ownership last. This is referred to as the *rotating ID*. When reset is asserted, the rotating ID is reset to three in all of the processors. This means that they all think that processor (i.e., agent) three owned the signal group last and therefore agent zero should get it next (if it asks for it). The sequence in which the processors gain ownership (if all of the processors were asking for ownership when reset was deasserted) is 0, 1, 2, 3, 0, 1, etc.

Busy/Idle State

In addition to the rotating ID, each processor must also keep track of whether the last processor that gained ownership of the request signal group retained ownership or has released it (and therefore none of them currently owns it). When the last owner retains ownership, the ownership state is said to be *busy*. If the previous owner surrendered ownership and none of them currently owns the request signal group, the ownership state is said to be *idle*. Each of the processors maintains an internal state indicator to indicate whether the bus ownership state is currently busy or idle.

It is incorrect to think of busy as meaning that one of the processors currently owns the request signal group and is using it. Rather, it only means that the processor has retained ownership—it may or may not currently be using the request signal group.

Bus Parking

The processor may retain ownership after completing a transaction in case it may need the request signal group again in the future (note that the Pentium Pro processor does this). This is referred to as parking ownership on yourself. If a processor is successful at retaining ownership until such time as it may

Chapter 10: Obtaining Bus Ownership

require the signal group again, parking provides faster access to the request signal group (because you don't have to issue a request for the current symmetric owner to surrender ownership). More information on parking can be found in "Bus Parking Revisited" on page 210.

Be Fair!

When a processor parks ownership on itself, it may retain ownership until:

- it must start a new transaction or
- another processor request ownership,

whichever occurs first. In other words, be fair to the other processors—don't hog the bus.

What Signal Group are You Arbitrating For?

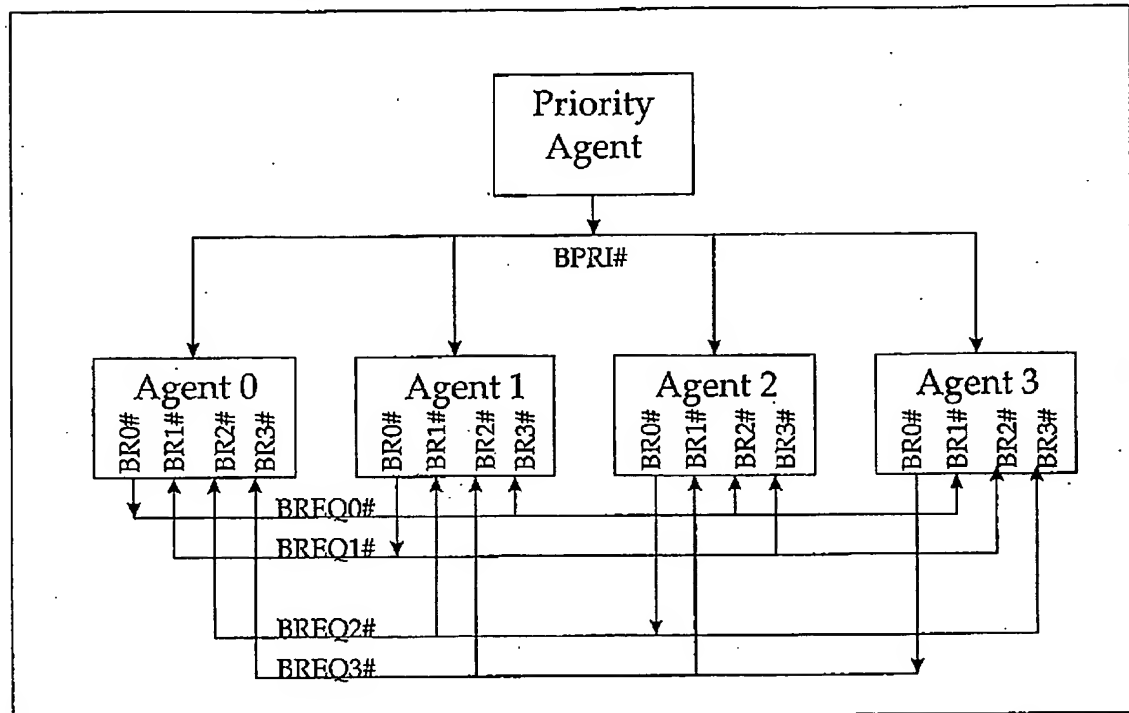
When an agent wins the arbitration, it takes ownership of the request signal group and uses it to issue a transaction request to the other agents that reside on the processor bus. A detailed account of the request phase of the transaction can be found in the chapter entitled "The Request and Error Phases" on page 241.

Requesting Ownership

Refer to Figure 10-2 on page 206. The temptation is great to think that agent 0 uses its BR0# output to request ownership, agent 1 uses its BR1# output, etc., but it's not true. When any processor wants to issue a new transaction request, it uses its bus request signal, BR0#, to request ownership. BR[3:1]# are inputs that are sampled to see if anyone else is also requesting ownership at the same time. If agent 0 is issuing a request, the BREQ0# signal line is asserted. If agent 1 is issuing a request, the BREQ1# signal line is asserted, etc. Each of the processors knows which of the BREQn# signal lines belongs to it and which belongs to each of the other processors.

Pentium Pro Processor System Architecture

Figure 10-2: Symmetric Agent Arbitration Signals



Example of One Symmetric Agent Requesting Ownership

Figure 10-3 on page 207 illustrates processor 0 requesting ownership. The following list describes the figure.

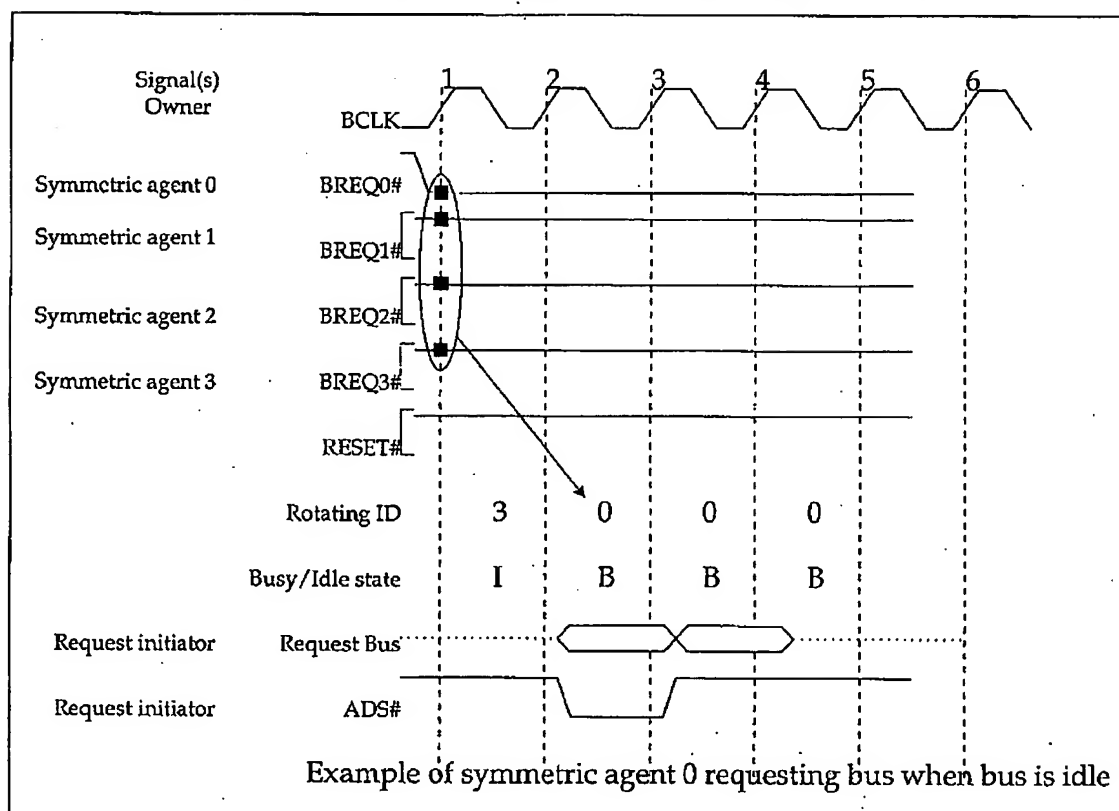
1. Prior to the rising-edge of clock one, none of the processors was requesting ownership (e.g., immediately after reset removal). The bus ownership state within each of them is "idle" and the last one that had ownership was agent 3 (note that this is the condition immediately after reset is removed).
2. During the clock immediately prior to clock 1, agent 0 needs to perform a transaction and asserts BREQ0# to indicate this.
3. On the rising-edge of clock one, agents 1, 2, and 3 detect that agent 0 is requesting ownership. A change in the state of the BREQn# signals from none asserted in one clock to one or more asserted in the next is an arbitration event.
4. One clock later, during clock two, all of the processors change the state of their rotating ID indicator to indicate that agent 0 is now the owner. They also change the state of their ownership state indicators to indicate that the ownership state is now busy—in other words, one of them now owns the portion of bus required to issue a new transaction. Note that the rotating ID

Chapter 10: Obtaining Bus Ownership

and ownership state indicators are strictly internal to the processors. They are not presented on any output pins.

5. In clock two, agent 0 takes ownership of the signals required to issue a new transaction and drives out its request and asserts ADS# (Address Strobe) to indicate that a new transaction is being issued to the bus. It leaves BREQ0# asserted either because it wants to issue another transaction request immediately after this one, or in case it needs the bus again in the future (see "Bus Parking Revisited" on page 210).

Figure 10-3: Example of One Processor Requesting Ownership



Example of Two Symmetric Agents Requesting Ownership

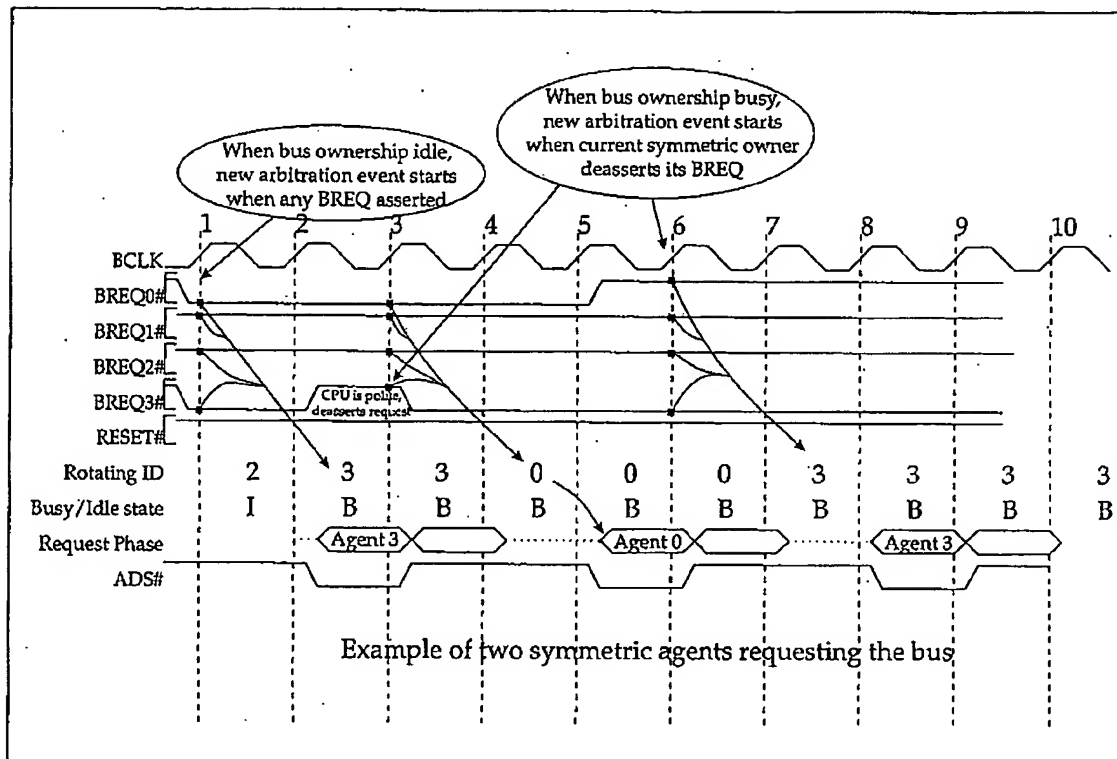
Figure 10-4 on page 209 illustrates a case where symmetric agents 0 and 3 are both requesting ownership. Arbitration events (where the symmetric agents must decide which of them ownership will pass to) occur on the rising-edge of clocks 1, 3, and 6. The following list describes the figure.

Pentium Pro Processor System Architecture

1. Prior to clock 1, none of the symmetric agents were requesting ownership. The previous owner was agent 2 and the ownership state is currently idle (i.e., none of the symmetric agents has ownership). *Please note that the Pentium Pro processor always leaves its BREQ asserted (in case it may need to issue another request in the future) when it initiates a transaction request. In other words, it uses parking.*
2. On clock 1, requests from agents 0 and 3 are both seen. *This is an arbitration event.*
3. In clock 2, all symmetric agents change the rotating ID from 2 to 3 and the ownership state from idle to busy. Agent 3 takes ownership of the request signal group, issues a request and asserts ADS# (Address Strobe) to indicate that a new transaction request is being issued. As it's issuing the request, agent 3, recognizing that one (or more) other symmetric agents are also requesting ownership, does the polite thing—it deasserts BREQ3# to relinquish ownership.
4. On clock 3, an *arbitration event* is detected (agent 3 has relinquished ownership). All symmetric agents see that only agent 0 is requesting ownership. Having been polite to agent 0, agent 3 reasserts BREQ3# to request ownership again (because it has another transaction to issue).
5. One clock later (in clock 4), all symmetric agents change their rotating ID to 0 and the ownership state stays busy (there was a hand off of ownership from one to the other). Before taking ownership, however, agent 0 must wait for agent 3 to completely cease driving the request signal group.
6. In clock 5, agent 0 begins to drive its request onto the request signal group and asserts ADS#. Having detected BREQ3# asserted, it also deasserts BREQ0# to hand ownership back to agent 3.
7. In clock 6, an *arbitration event* is detected (agent 0 has relinquished ownership). All symmetric agents see that only agent 3 is requesting ownership.
8. In clock 7, all symmetric agents change their rotating ID to 3 and the ownership state stays busy. Before taking ownership, however, agent 3 must wait for agent 0 to completely cease driving the request signal group.
9. In clock 8, agent 3 begins to drive its request onto the request signal group and asserts ADS#. It keeps BREQ3# asserted either because it has another transaction request to issue or to park ownership on itself in case it needs in the future.

Chapter 10: Obtaining Bus Ownership

Figure 10-4: Example of Two Symmetric Agents Requesting Ownership



Definition of an Arbitration Event

An arbitration event is defined as the passing of ownership (of the request signal group) from one symmetric agent to another. This occurs under the following circumstances:

- When none of the $BREQ_n\#$ lines are asserted during one clock and then one or more are seen asserted in the next clock. An example of this can be seen in Figure 10-3 on page 207 during the clock that precedes clock one and clock one.
- When the current symmetric owner of the request signal group relinquishes ownership by deasserting its $BREQ_n\#$ signal and one or more of the other $BREQ_n\#$ lines have previously been asserted.

In either case, the symmetric agents must collectively decide (based on who owned it last and therefore who's next in the rotation) which of them will assume ownership of the request signal group in the next clock.

Pentium Pro Processor System Architecture

Once BREQn# Asserted, Keep Asserted Until Ownership Attained

Once a symmetric agent asserts its BREQn# signal, it must keep it asserted until it attains ownership (see "Other Guys are Very Polite" on page 256 for a description of the only exception). Once ownership is attained, the agent may or may not generate a transaction request (usually it will). If the agent no longer needs to generate the transaction request for which it originally asserted its BREQn#, it can deassert its BREQn# (without generating a transaction request) once it has attained ownership.

Example Case Where Transaction Cancelled Before Started

An example would be the case where the processor has a modified line sitting in one of its writeback buffers waiting to be cast back to memory (to make room for a new line being read into the L2 cache). The processor had asserted its BREQn# to request ownership to do the cast out. Before it acquired ownership, however, another agent attained ownership and issued a read for the same line. The processor waiting to writeback the line snoops its writeback buffers, asserts HITM# (hit on modified line) and supplies the modified line directly to the agent that issued the read request. There is no longer a need to perform the writeback, but the processor must keep its BREQn# asserted until it attains ownership. It then deasserts its BREQn#. The other symmetric agents will see that it is yielding ownership on the next clock and, if any other symmetric agents have their BREQs asserted, ownership passes to the next in the rotation.

Bus Parking Revisited

A symmetric agent can park ownership of the request signal group on itself if it so desires (in other words, parking is optional). It does this by not deasserting its BREQn# signal when it issues a transaction request. This occurs under two circumstances:

1. Symmetric agent issues a transaction request and has another to issue immediately after the first. It can retain ownership of the request signal group by keeping its BREQn# asserted when issuing the first transaction request. Note that if any other symmetric agents are requesting ownership, this isn't very polite to them.

Chapter 10: Obtaining Bus Ownership

2. Symmetric agent issues a transaction request and keeps its BREQn# asserted in case it needs to issue a request in the future. It can retain ownership until either:
 - it needs to issue a new transaction request, or
 - until it detects another symmetric agent's BREQn# line asserted.

Most of the time, *the Pentium Pro processor uses method 2*. There are some cases where the processor must prevent any other agent from gaining bus ownership in between two of its transactions, however, and then it uses method 1 (for more information see "Locking—Shared Resource Acquisition" on page 221).

Examples of symmetric ownership parking are illustrated in clock 2 of Figure 10-3 on page 207 and clock 8 of Figure 10-4 on page 209.

Priority Agent Arbitration—Despotism

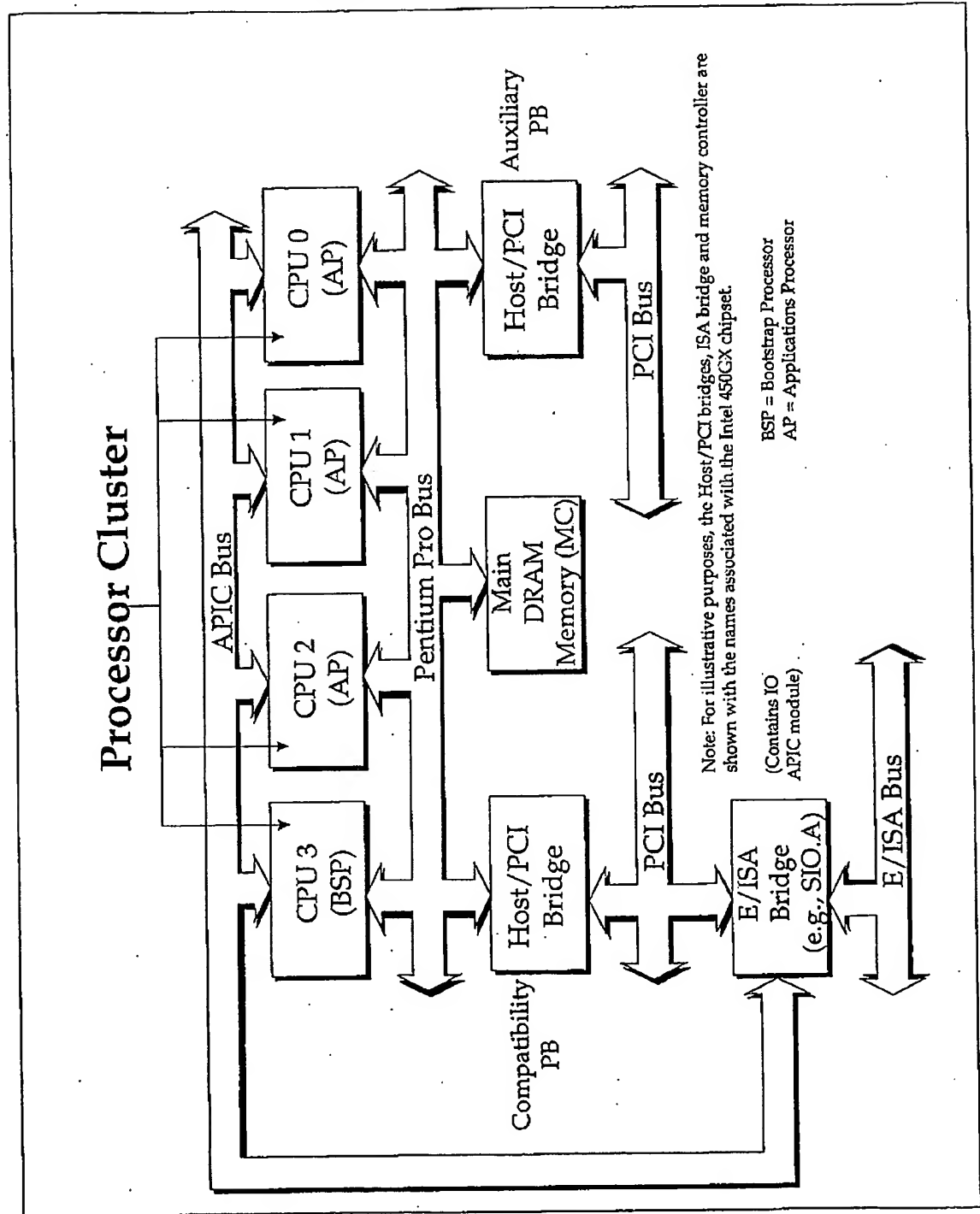
Example Priority Agents

While the symmetric agents are very polite to each other, the system may include one or more agents that play by different rules. They are referred to as priority agents. Refer to Figure 10-5 on page 212. As an example, in the 450GX chipset, both of the PBs (i.e., the host/PCI bridges) are priority agents. They use the BPRI# signal (note that there is only one BPRI# signal) to request ownership when they need to issue a transaction request on the processor bus. The PBs act as the surrogate processor bus request agent when a PCI master (or an ISA or EISA master) requires access to main memory.

Only one device is permitted to assert BPRI# at a time. In the case where there are multiple priority agents, there must therefore be some method for the priority agents to arbitrate amongst themselves to determine which of them gets to use BPRI# to request ownership (if more than one of them needs to issue a transaction request at the same time). In the case of the 450GX's two PBs, they use two signals, IOREQ# and IOGNT#, to decide which of them gets to use BPRI#. In the case where both of them need to issue a transaction request, the compatibility PB wins.

Pentium Pro Processor System Architecture

Figure 10-5: System Block Diagram



Chapter 10: Obtaining Bus Ownership

Priority Agent Beats Symmetric Agents, Unless...

When a priority agent is requesting ownership at the same time that one or more of the symmetric agents are also requesting ownership, the priority agent wins.

The only case where the priority agent will be unsuccessful in winning ownership of the bus is the case where a symmetric agent has already acquired ownership and has asserted the LOCK# signal. This prevents the priority agent from acquiring ownership until the symmetric agent deasserts the LOCK# signal. The reasons why a symmetric agent might assert LOCK# are covered in the section entitled "Locking—Shared Resource Acquisition" on page 221. The priority agent must deal with the cases described in Table 10-1 on page 213.

Table 10-1: Possible Cases Involving Priority Agent Arbitration

Case	Resulting Actions
A symmetric agent initiates a transaction request in the same clock that the priority agent asserts BPRI#, but does not assert LOCK#.	In this case, the priority agent assumes ownership after the symmetric agent finishes delivery of its transaction request. This will be 3 clocks after BPRI# assertion.
A symmetric agent initiates a transaction request and asserts LOCK# in the same clock that the priority agent is asserting BPRI#.	The priority agent cannot assume ownership until the symmetric agent deasserts LOCK#.
A symmetric agent has acquired ownership on the same rising-edge of the clock that BPRI# is sampled asserted. In this case, the symmetric agent proceeds with its transaction request and may or may not assert LOCK#.	<ul style="list-style-type: none">• If LOCK# is asserted, the priority agent doesn't acquire ownership until LOCK# is deasserted by the symmetric agent.• If LOCK# isn't asserted, the priority agent acquires ownership as soon as the symmetric agent completes issuing its transaction request. This will be 2 clocks after BPRI# is asserted.

Pentium Pro Processor System Architecture

Using Simple Approach, Priority Agent Suffers Penalty

Refer to Figure 10-6 on page 216. A priority agent may be designed in such a manner that it doesn't check to see if a symmetric agent has started a transaction request (in other words, it doesn't check the state of the ADS# signal) in order to determine when (and if) it can take ownership of the request signal group. Rather, it checks in the two clocks immediately following its assertion of BPRI# to see if LOCK# is asserted. If LOCK# is sampled asserted on the rising-edge of either of the two clocks immediately after BPRI# is asserted, then a symmetric agent had already asserted LOCK# and the priority agent can't take ownership until LOCK# is deasserted. If LOCK# is sampled deasserted during both of these two clocks, however, one of three conditions is true (but the priority agent doesn't know which case is true):

1. No symmetric agent has initiated a transaction request during these two clocks and LOCK# is not being held asserted by a symmetric agent that issued an earlier transaction request.
2. A symmetric agent started a transaction request on the same clock that BPRI# was driven asserted, but did not assert LOCK#.
3. A symmetric agent started a transaction request on the clock after BPRI# was asserted, but did not assert LOCK#.

In any of these cases, the priority agent has gained ownership. However, because it doesn't check ADS# to determine which of the three cases is true, it must assume the worst case—case number 3. In this case, the symmetric agent has sampled BPRI# asserted on the same rising-edge of the clock that the symmetric agent initiated its transaction request, it does not assert LOCK#, and it will therefore honor the BPRI# assertion. The priority agent cannot assume ownership, however, until 3 clocks after the symmetric agent starts its transaction request. This is a *total of 4 clocks after BPRI# is asserted*.

1. An arbitration event occurs on clock 2 in Figure 10-6 on page 216 and agent 0 acquires ownership in clock 3 (BPRI# is not yet asserted, so agent 0 is not prevented from taking ownership). Also on clock 2, the priority agent asserts BPRI# to request ownership, but this isn't detected by agent 0 until clock 3, the clock in which it starts to drive out a transaction request. This means that agent 0 has successfully acquired ownership and will proceed with the issuance of its transaction request.
2. When agent 0 starts its request in clock 3, it keeps BREQ0# asserted in case it has another transaction to issue later. Also in clock 3, agent 1 asserts

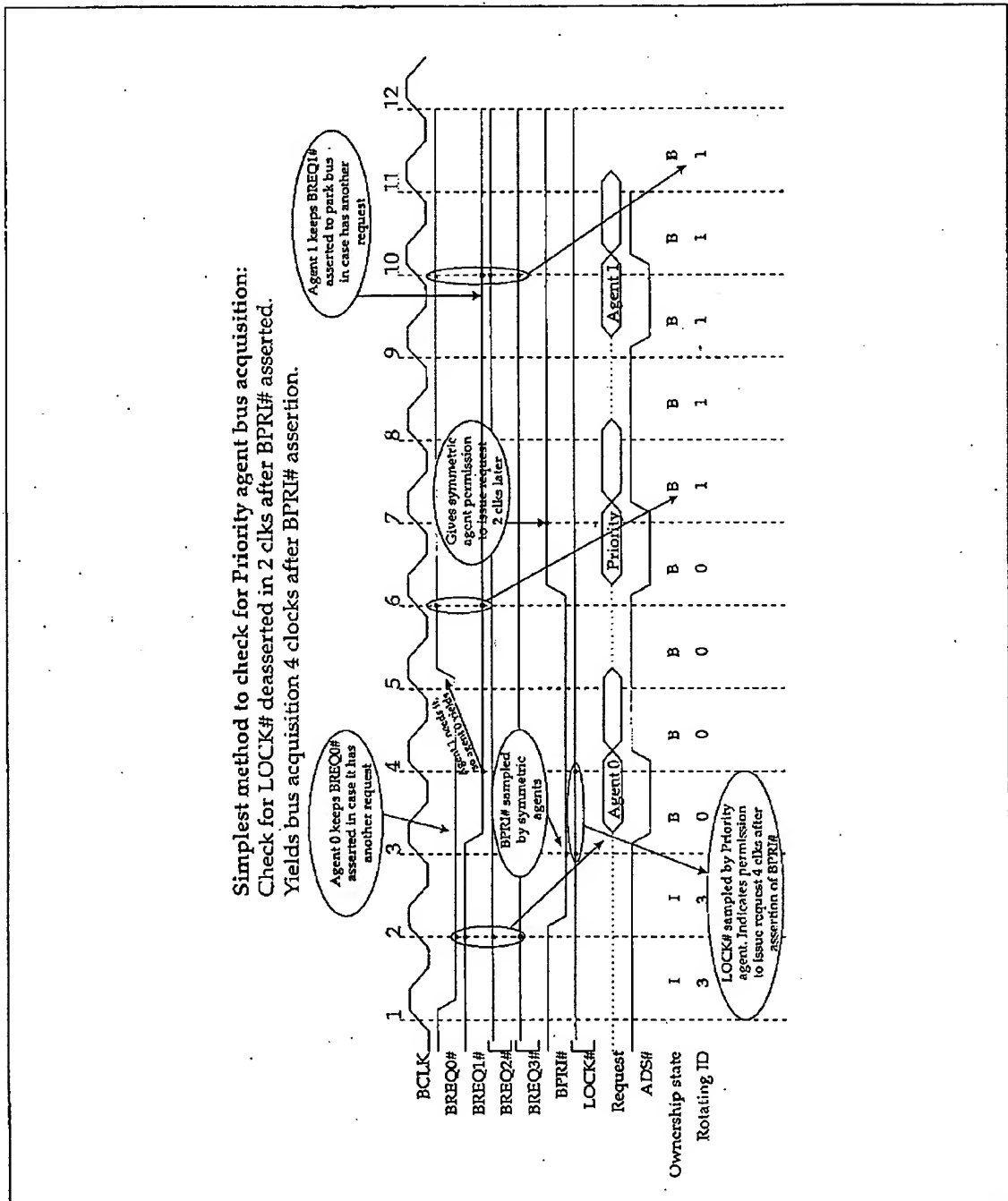
Chapter 10: Obtaining Bus Ownership

BREQ1# to request ownership, but it won't gain symmetric ownership until agent 0 relinquishes ownership.

3. The priority agent uses the simple approach to test for ownership acquisition. It samples LOCK# on clocks 3 and 4 to see if any symmetric agent has asserted it. In this case, it is sampled deasserted both times, indicating either that no symmetric agent is issuing a request, or one is, but has not asserted LOCK#. In either case, it means the priority agent will be the next owner of the request signal group.
4. Agent 0 samples BREQ1# asserted on clock 4 and relinquishes ownership in clock 5.
5. An arbitration event occurs on clock 6 when BREQ0# is sampled deasserted indicating that agent 0 has relinquished ownership to agent 1. In clock 7, ownership passes to agent 1.
6. On clock 6, 4 clocks after it asserted BPRI#, the priority agent takes ownership of the request signal group, initiates a transaction request, and deasserts BPRI# to let the next symmetric agent use the request signal group after it is done. Note that if none of the symmetric agents had their BREQn# lines asserted, the priority agent could have left BPRI# asserted to park ownership on itself in case it needed to issue another transaction request in the future. Since both agents 0 and 1 have asserted their respective BREQs, however, the priority agent is a gentleman and releases BPRI#.
7. On clock 7, symmetric ownership passes to agent 1. In addition, BPRI# is sampled deasserted by the symmetric agents, indicating that the next symmetric owner can start issuing a request 2 clocks later (on clock 9).
8. When agent 1 issues its transaction request on clock 9, it keeps BREQ1# asserted to park symmetric ownership on itself in case it needs to issue another transaction request later.

Pentium Pro Processor System Architecture

Figure 10-6: Simple Approach Results in Penalty



Chapter 10: Obtaining Bus Ownership

Smarter Priority Agent Gets Ownership Faster

The previous section demonstrated that a priority agent that only checks the state of the LOCK# signal to determine if and when it has attained ownership takes at least four clocks to attain ownership (longer if LOCK# is sampled asserted). This section describes how a priority agent that checks both ADS# and LOCK# can *attain ownership in 2 or 3, rather than 4 clocks*.

Table 10-1 on page 213 detailed the possibilities when a priority agent asserts BPRI# to request ownership. If the priority agent checks to see if a symmetric agent has started a transaction request in the same clock that it asserts BPRI# or the clock immediately following its assertion, it can decrease the latency in gaining ownership from 4 to 3 (Figure 10-8 on page 220) or 2 (Figure 10-7 on page 218) clocks.

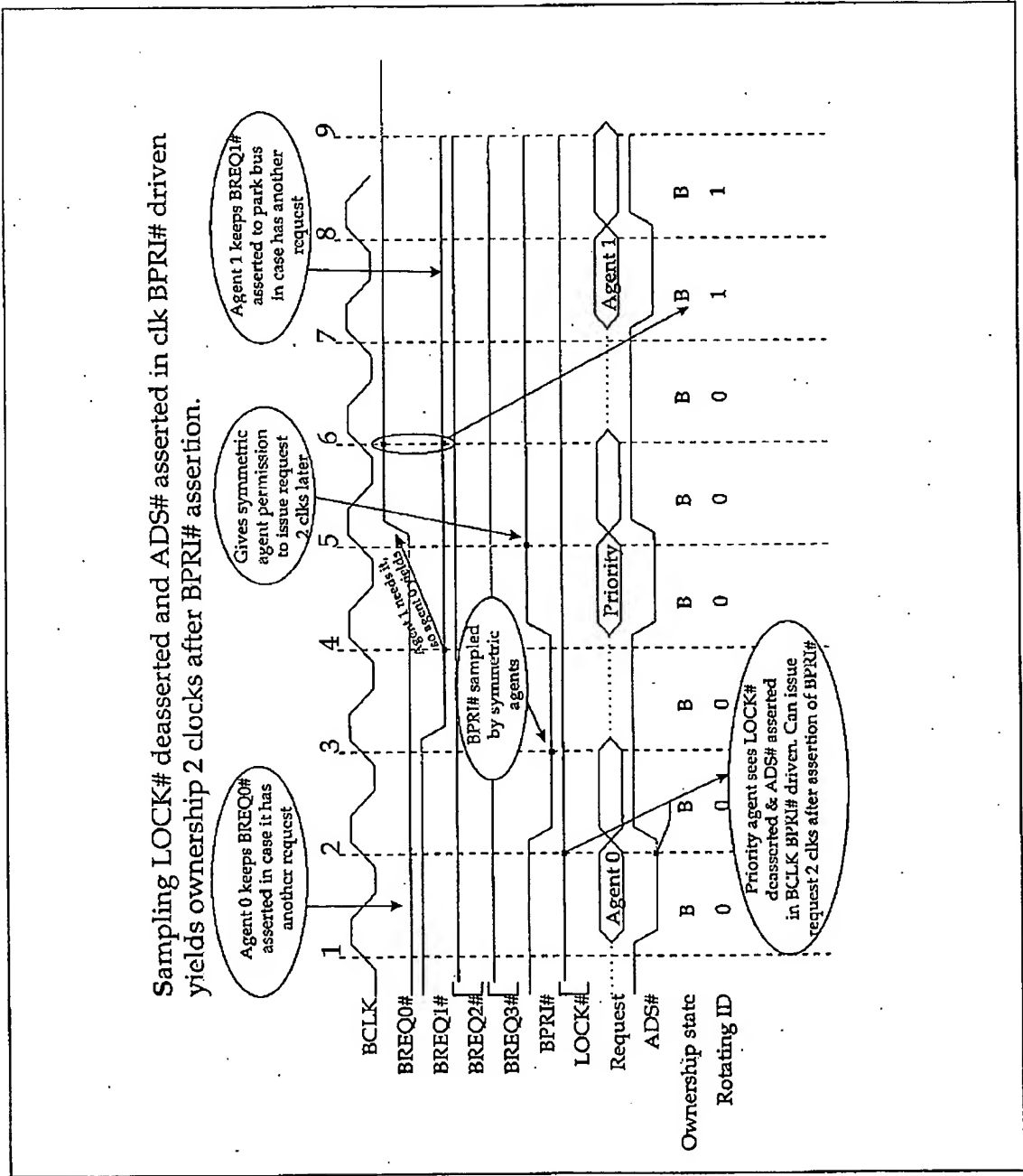
Ownership Attained in 2 BCLKs

If the priority agent samples ADS# asserted (indicating that a symmetric agent has initiated a transaction request) and LOCK# deasserted (but it isn't locking the request signal group) in the same clock that it asserts BPRI#, the priority agent can assume ownership in 2 clocks. Refer to Figure 10-7 on page 218.

1. On clock 2, the priority agent begins to assert BPRI#. At the same time, agent 0 initiates a transaction request and asserts ADS#. The priority agent samples ADS# asserted and LOCK# deasserted, indicating that a symmetric agent has initiated a transaction request but has not locked the request signal group.
2. All of the symmetric agents sample BPRI# asserted on clock 3, indicating that the priority agent will be the next owner of the request signal group.
3. On clock 4, after agent 0 has completed issuance of its transaction request, the priority agent initiates its transaction request. BREQ1# is also sampled asserted (by the other symmetric agents as well as the priority agent), indicating that symmetric agent 1 wishes to issue a transaction request. Being a good bus citizen, the priority agent deasserts BPRI# to yield the bus to the symmetric agent.
4. On clock 5, agent 0 deasserts BREQ0# to yield symmetric ownership to agent 1.
5. On clock 6, the symmetric agents detect BREQ0# deasserted and select the next symmetric owner, agent 1.
6. On clock 7, symmetric ownership passes to agent 1 and it initiates its transaction.

Pentium Pro Processor System Architecture

Figure 10-7: Example Where Priority Agent Attains Ownership in 2 Clocks



Chapter 10: Obtaining Bus Ownership

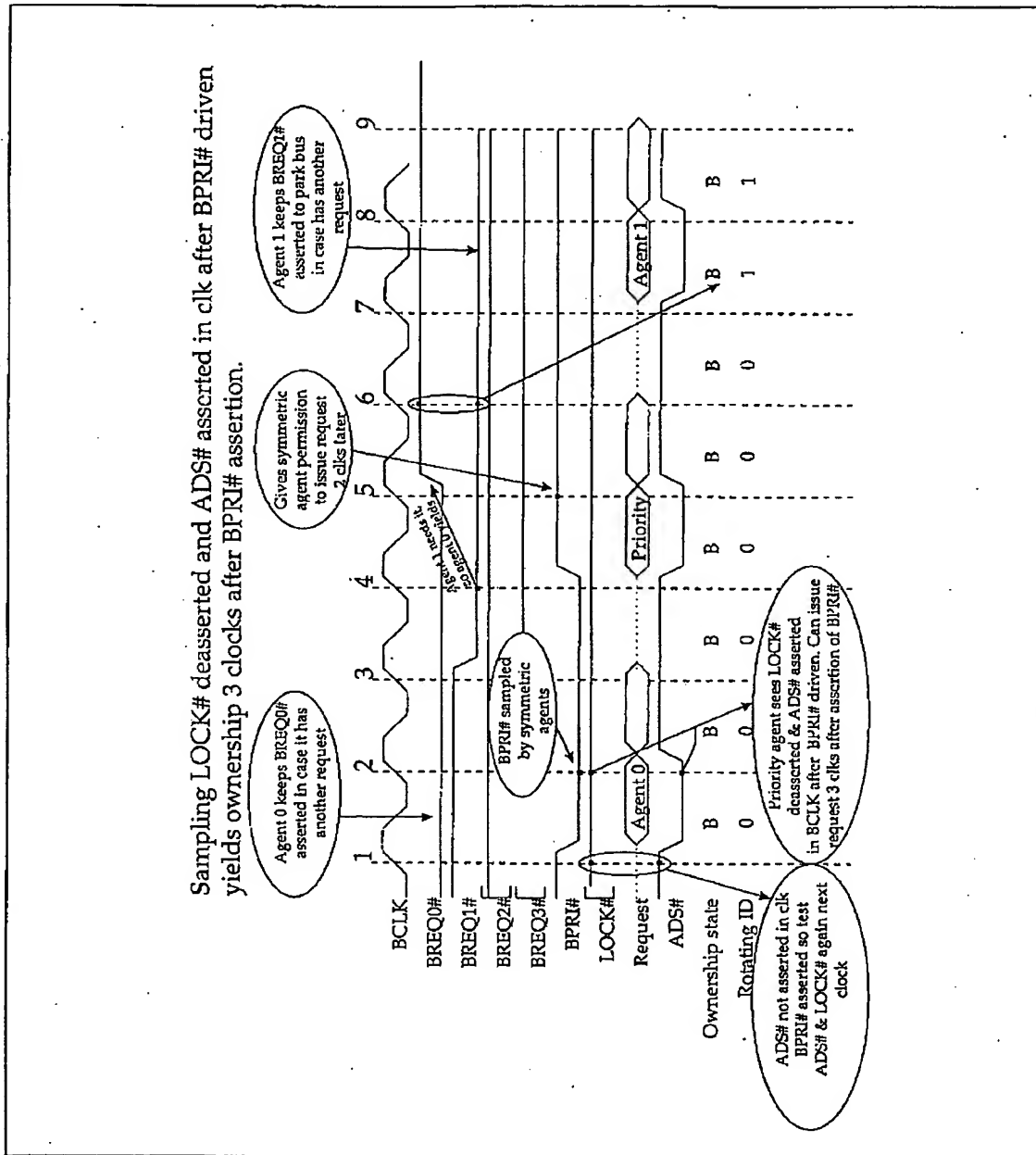
Ownership Attained in 3 BCLKs

If the priority agent samples ADS# asserted (indicating that a symmetric agent has initiated a transaction request) and LOCK# deasserted (but it isn't locking the request signal group) in the clock after it asserts BPRI#, the priority agent can assume ownership in 3 clocks. Refer to Figure 10-8 on page 220.

1. On clock 1, agent 0 initiates a transaction request and the priority agent asserts BPRI# to request ownership. ADS# is sampled deasserted, indicating that a symmetric agent didn't just start a transaction request. Check ADS# and LOCK# again on next clock.
2. On clock 2, ADS# sampled asserted, indicating that a symmetric agent has just initiated a transaction request. LOCK# is sampled deasserted, indicating that the symmetric agent didn't lock the request signal group.
3. The priority agent can assume ownership after the symmetric agent completes issuing its transaction request. This occurs on clock 4.

Pentium Pro Processor System Architecture

Figure 10-8: Example Where Priority Agent Attains Ownership in 3 Clocks



Chapter 10: Obtaining Bus Ownership

Be Fair to the Common People

The spec dictates that the priority agent must keep BPRI# deasserted for at least two clocks after its last deassertion of BPRI#. This opens a window that permits the symmetric agents to get ownership of the request signal group if any of them need to issue a transaction request.

Priority Agent Parking

The specification says that the priority agent can keep BPRI# asserted when it asserts ADS# and initiates its final (or only) transaction request. The exact wording is “provided it can guarantee forward progress of the symmetric agents.” In other words, **be fair to the common people**. If the priority agent has parked ownership on itself by keeping it BPRI# asserted, it can retain ownership until:

- it has to initiate another transaction request, or
- it samples any of the BREQn# lines asserted, indicating that one or more of the symmetric agents require access to the request signal group,

whichever comes first.

Locking—Shared Resource Acquisition

The previous section, “Priority Agent Arbitration—Despotism,” illustrated how the assertion of LOCK# by a symmetric agent prevents the priority agent from getting ownership. This section describes the reasons why a symmetric agent might need to perform a series of transactions without fear of any other agent performing an access in between its own transactions.

Shared Resource Concept

Assume that the OS sets aside an area of memory to be used by tasks executing on multiple processors (or even by different tasks executed by the same processor) as a shared memory buffer. It is intended to be used as follows:

1. Before using the buffer (i.e., reading from or writing to it), a task must first test a flag to ensure that the buffer isn’t currently owned by another task. If the buffer is currently unavailable, the task wishing to gain ownership should periodically check back to see when it becomes available.
2. When the flag indicates that the buffer is available, the task should set the flag to indicate that it now has exclusive ownership of the buffer. It will

Pentium Pro Processor System Architecture

then be seen as unavailable if any other task should attempt to gain ownership of it.

3. Having gained exclusive ownership of the buffer, the task can now read and write the buffer.
4. If the buffer is in an area of memory designated as WT, WC, or UC memory (refer to "Rules of Conduct" on page 119), writes are absorbed into the processor's posted write buffers. *These buffers are not snooped* when other agents access memory. In this case, when the task is done using the buffer, it should ensure that all of its updates (i.e., memory writes) have been flushed all the way to memory.
5. After ensuring that the buffer has received all updates, the task should release ownership of the buffer so it can be used by other tasks.

Testing Availability and Gaining Ownership of Shared Resources

The OS typically uses a memory location (or series of memory locations) as the flag indicating the availability or unavailability of a particular shared resource. This is referred to as a *memory semaphore*. It is used as follows:

1. Before using the buffer (i.e., reading from or writing to it), a task reads the buffer's semaphore to ensure that the buffer isn't currently owned by another task. If the buffer is currently unavailable (usually indicated by a non-zero semaphore value), the task wishing to gain ownership should periodically check back to see when it becomes available.
2. When the flag indicates that the buffer is available (semaphore contains a zero value), the task writes a non-zero value into the semaphore to indicate that it now has exclusive ownership of the buffer. The buffer will then be unavailable if any other task should test the semaphore.
3. Having gained exclusive ownership of the buffer, the task can now read and write the buffer.
4. If the memory buffer area is designated as WC, WT, or UC memory, when the task is done using the buffer, it should ensure that any buffer updates (i.e., memory writes) have been flushed all the way to memory.
5. After ensuring that the buffer has received all updates, the task releases ownership of the buffer (by clearing the semaphore to zero) so it can be used by other tasks.

Race Condition Can Present Problem

Consider the following possibility:

1. The task executing on processor 0 reads the semaphore to determine the

Chapter 10: Obtaining Bus Ownership

buffer's availability.

2. The task tests the semaphore's value and determines that the buffer is available (semaphore value is zero).
3. After the task on processor 0 has completed the memory read to obtain and test the semaphore value, a task executing on processor 1 has initiated a memory read request to test the state of the same semaphore. It completes the read and begins testing the value.
4. The processor 0 task initiates a memory write request to update the semaphore to a non-zero value to mark the shared buffer as unavailable. After it completes the write, it considers itself the sole owner of the buffer.
5. The processor 1 task also determined the buffer is available and it now performs a memory write request to update the semaphore to a non-zero value to mark the shared buffer as unavailable. It completes the write and it also now considers itself the sole owner of the buffer.

Two tasks executing on two separate processors now each believe that it has exclusive ownership of the buffer.

Guaranteeing Atomicity of Read/Modify/Write

This problem came about because processor 1 was able to read the semaphore immediately after processor 0 read it. The two processors were in a race condition. Processor 0 wrote to it, followed by processor 1 writing to it. The tasks on the two processors each ended up believing it had sole ownership of the buffer.

The problem can be prevented if processor 0 could prevent other initiator's from using the bus from the time it initiates its read until the time it updates the semaphore to a non-zero value. In other words, it should lock the bus while it performs the read/modify/write (frequently referred to as a RMW) of the semaphore.

To do this, the programmer uses special instructions to perform the RMW operation. When using these instructions, the processor (refer to Figure 10-9 on page 224):

1. asserts the LOCK# signal when it initiates the memory read, keeps LOCK# asserted while it performs the internal semaphore test, and performs the memory write to update the semaphore before releasing the LOCK# signal. The assertion of LOCK# prevents the priority agent from obtaining bus ownership during this period.
2. also keeps its BREQn# output asserted throughout this period to keep any of the other processors from obtaining ownership of the request signal group.

Chapter 10: Obtaining Bus Ownership

LOCK Instruction Prefix

The following instructions may be prefixed with the lock prefix to force the assertion of LOCK# for the duration of the read and write:

- bit test and modify instructions: BTS, BTR, and BTC.
- exchange instructions: XADD, CMPXCHG, and CMPXCHGB.
- XCHG instruction doesn't require the lock prefix to assert LOCK#.
- the following single-operand arithmetic and logical operations: INC, DEC, NOT, and NEG.
- the following two-operand arithmetic and logical operations: ADD, ADC, SUB, SBB, AND, OR, and XOR.

Processor Automatically Asserts LOCK# for Some Operations

The processor automatically asserts LOCK# under the following circumstances:

- execution of the XCHG instruction when it accesses memory.
- setting the Busy flag of a TSS (task state segment). For information on the TSS and Busy flag, refer to the MindShare book entitled *Protected Mode Software Architecture* (published by Addison-Wesley).
- when the processor reads a segment descriptor from memory, it asserts LOCK# during the read, tests the state of the descriptor's Accessed bit and, if clear, performs the memory write to set the bit in the descriptor in memory before releasing LOCK#.
- when updating the Accessed and/or Dirty bits in page directory and page table entries.

Use Locked RMW to Obtain and Give Up Semaphore Ownership

The programmer should always:

- use a locked RMW instruction to obtain ownership of the semaphore, and
- use the locked instruction to release ownership (in other words, perform a locked RMW to change the semaphore back to zero).

Locked instructions are *serializing, synchronizing* operations. Remember that the Pentium Pro processor performs out-of-order execution. This means that if the programmer used an unlocked RMW to release the semaphore, it could be executed before all of the code that precedes it has been executed. Using a locked RMW instruction ensures that the processor will execute all instructions before

Pentium Pro Processor System Architecture

the locked instruction prior to executing it (i.e., it is serializing). In addition, the locked instruction is synchronizing—it forces all posted writes within the processor to be flushed to external memory before executing the next instruction. This ensures that the buffer has received all updates before it is released.

Duration of Locked Transaction Series

The number of transactions necessary to perform the RMW on a semaphore depends on the placement in memory and the size of the semaphore. The Pentium Pro processor has a 64-bit data bus (eight data paths) and addresses memory on quadword boundaries. Using the address bus, it can therefore identify a block of eight memory locations aligned on a quadword boundary and use its byte enables to identify which bytes are to be read or written within the addressed quadword. This being the case, it can read or write a semaphore using one transaction under the following circumstances:

- semaphore is one byte wide.
- semaphore is one word wide (16-bits) and is aligned on a word address boundary.
- semaphore is one dword wide (32-bits) and is aligned on a dword address boundary.
- semaphore is one quadword wide (64-bits) and is aligned on a quadword address boundary.

Earlier x86 processors had 32- rather than 64-bit data buses, so software written to be executed on any x86 processor can only guarantee a single access read or write under the following circumstances:

- semaphore is one byte wide.
- semaphore is one word wide (16-bits) and is wholly-contained within in dword.
- semaphore is one dword wide (32-bits) and is aligned on a dword address boundary.

When performing a RMW operation on a semaphore that falls completely within one quadword, the Pentium Pro processor only has to perform one read and one write transaction (a total of two transactions) to accomplish the RMW.

However, if a semaphore value is set up in memory starting on a misaligned address boundary (for example, a dword semaphore that straddles two quadwords), the processor must perform two reads to get the semaphore and two writes to update it (a total of four transactions). This is inefficient from the processor's standpoint and also from a system perspective (because the bus is

Chapter 10: Obtaining Bus Ownership

locked for a longer period). When a semaphore is split across boundaries, the processor will assert LOCK# when it initiates the first memory read and keep it asserted for the four resulting transactions. In addition, however, it will also assert SPLCK# (Split Lock) to inform the addressed memory (or an L3 cache) that the currently-addressed quadword and the next should be locked. Refer to Table 11-11 on page 252.

The worst-case alignment scenarios occur when a semaphore straddles cache line or page boundaries. In the case of a semaphore that straddles cache line boundaries where neither line is currently in the processor's caches, the processor must fetch both 32-byte lines from memory to obtain the semaphore and must lock the bus for this entire period. In the case of a semaphore that straddles 4KB page boundaries where neither page is currently in memory, two complete pages, 8KB of information, must be read into memory from mass storage before the processor can read the semaphore.

Locking a Cache Line

When the processor begins the locked RMW operation, there are several possible cases:

- the semaphore value isn't in the cache.
- the semaphore value is in the cache in the E state.
- the semaphore value is in the cache in the S state.
- the semaphore value is in the cache in the M state.

Intel states that the Pentium Pro processor implements *cache line locking* in areas of memory designated as *WB memory*. It should be noted that Intel provides almost no information on how this works, but the author is as certain as can be that the following descriptions are accurate. Some important points to keep in mind are covered in the sections that follow.

Advantage of Cache Line Locking

Bus locking is inefficient—for the duration of a processor's RMW operation, no other bus agent can initiate a new transaction. If there are a lot of RMW operations being performed by the processors and/or priority agents, this can severely degrade system performance. If a semaphore is cached by the Pentium Pro processor, the RMW operation can be performed without locking the bus.

Pentium Pro Processor System Architecture

New Directory Bit—Cache Line Locked

Intel states that a cache line can be locked. This implies that there is a lock bit in each L1 data cache directory entry that is used for this purpose.

Read and Invalidate Transaction (RWITM, or Kill)

In the event of a race condition, you don't want multiple processors that have a cached copy of the line (in the S state) to be testing the same cached semaphore simultaneously. Therefore, before performing a locked RMW operation in the cache, a processor must first gain exclusive ownership of the line. This implies that you have to kill everyone else's copy of the line. Intel has included a special transaction type, read and invalidate, specifically for this purpose. The PowerPC 60x bus has two transaction types that perform a similar role: kill and RWITM (read with intent to modify).

Line in E or M State

A processor that has a copy of the line in the E or M state has the only copy of the line and therefore doesn't have to worry about another processor testing a copy of the semaphore in its cache at the same time that it is doing so. However, it is possible that another processor attempting to read the semaphore will experience a cache miss and initiate a read and invalidate transaction to obtain an exclusive copy of the line on which to perform its RMW. The processor with the E or M copy must prevent the other processor from obtaining the line that contains the semaphore until it has finished its own RMW operation on the semaphore within the line. This is accomplished by marking the line locked in the cache when the read of the internal RMW is initiated. When the snoop of the other processor's read and invalidate transaction results in a hit on a locked line, the processor delays presentation of the snoop result to the other processor until its RMW has been completed. It then indicates the state of the line after the RMW. This could be either I or M:

1. It would be I if, when read from the line, the semaphore had been set to a non-zero value by another task at an earlier time. In this case, the processor doesn't update the line. Rather, it invalidates the line and indicates a snoop miss.
2. It would be M (snoop hit on M copy) in two cases:
 - The programmer didn't update the semaphore because it was already set, but some other item in the line had previously been updated (in other words, the line was in the M state before the RMW was initiated).
 - When read from the M line, the semaphore was zero, so the programmer wrote to the semaphore to update it and the line stayed in the M state.

Chapter 10: Obtaining Bus Ownership

Semaphore Not in Processor's L1 or L2 Cache

If the semaphore is not in the L1 data or the L2 cache when the attempt is made to read it, it results in a cache miss. The read request is submitted to the external bus unit and the processor initiates a read and invalidate transaction (has the effect of killing copies of the line in other processors' caches) to obtain the 32 byte line from memory. The read and invalidate transaction is used by the processor in cases when it is reading data from memory with the intent to modify it when the data has been obtained. This implies that any other processor that has a copy of the line in the E or S state should kill its copy of the line. If another processor has a copy of the line in the M state, it should source the line directly to the requesting processor and then kill its copy. When the line has been read from memory or from another processor's cache (in the case of a hit on an M line), the processor marks it locked in the data cache. It then performs the RMW operation. The line is unlocked when the RMW operation has been completed.

During this processor's RMW operation, another processor may initiate a read or a RMW that misses its cache, resulting in a read or a read and invalidate transaction for the same line. A snoop is performed in this processor's cache and hits on the locked line. The snoop result to the other processor is delayed (stretching the other processor's snoop phase) until the processor's RMW has been completed. The snoop result is then delivered to the other processor. Either the semaphore wasn't updated (and the line therefore wasn't modified), or it was (and the line was marked modified). If the line was modified, the line is provided directly to the other processor from this processor's cache (and is invalidated if the other processor's transaction type is the read and invalidate). Otherwise, the other processor obtains the line from memory. If the other processor's transaction is a read (rather than a read and invalidate), the line is marked S in this processor's cache. If the other processor's transaction was a read and invalidate, the line is invalidated in this processor's cache.

In the case where two processors are in a race condition where they are both initiating RMW operations that miss their caches, they both initiate read and invalidate transactions on the bus. Since they can't both initiate a transaction simultaneously, however, one wins bus ownership first and initiates its read and invalidate. The second processor then initiates its read and invalidate transaction. The first processor obtains the line from memory and marks it locked while it performs the RMW. When the second processor's read and invalidate transaction reaches its snoop phase, the first processor snoops its cache and hits on the locked line. It delays the delivery of the snoop result until it completes its RMW operation and then delivers the snoop result to the second processor. If the line is not modified, it indicates a miss to the second processor and invalidates its copy. On the other hand, if the RMW operation modified the line, it

Pentium Pro Processor System Architecture

indicates a hit on a modified line, sources the line directly to the second processor, and invalidates its copy of the line. The second processor now has a copy of the line and can perform its RMW operation.

Semaphore in Cache in E State

If the read of the RMW hits on an E copy of the line in the data or L2 cache, no other processor has a copy of the line in its cache. The line is marked locked in the cache (in case another processor tries to read the line during the RMW operation) and the RMW operation is then performed on the semaphore within line. The lock is then removed.

Semaphore in Cache in S State

If the read of the RMW hits on an S copy of the line in the data or L2 cache, at least one other processor has a copy of the line. Before this processor can perform the RMW, it must first gain exclusive ownership of the line by killing copies in the caches of other processors. It does this by issuing a read and invalidate transaction for 0 bytes of data (this is a kill). Any other processor that has a copy of the line must kill its copy. The line is then marked locked in the cache while the RMW operation is performed, after which it is unlocked.

Semaphore in Cache in M State

If the read of the RMW hits on an M copy of the line in the data or L2 cache, no other processor has a copy of the line in its cache. The line is marked locked in the cache (in case another processor tries to read the line during the RMW operation) and the RMW operation is then performed on the semaphore within the line. The lock is then removed.

Blocking New Requests—Stop! I'm Full!

The section entitled "Transaction Tracking" on page 199 introduced the concept of transaction tracking and the IOQ (in-order queue). Each agent has an IOQ that it uses to keep track of each transaction that is currently outstanding on the bus. The depth of an agent's IOQ is device-specific. The Pentium Pro processor has a selectable queue depth of either one or eight. The queue depths of the 450GX chipset agents is also selectable as one or eight. The queue depth of the 440FX chipset is four.

When the maximum number of transactions that a device can track are currently outstanding on the bus at various stages of completion, the agent cannot

Chapter 10: Obtaining Bus Ownership

permit any agent to initiate a new transaction. If a new transaction were initiated, the agent would be incapable of tracking it and consequently would lose track of all activity on the bus.

For this reason, agents must have the ability to throttle the ability of other agents to initiate new transactions. That is the purpose of the BNR# (Block Next Request) signal. BNR# could also be used by a debug tool to create a controlled situation where no additional transactions can be issued to the bus until the current transaction has been completed. In other words, transactions could be single-stepped onto the bus to simplify the debug process.

BNR# is Shared Signal

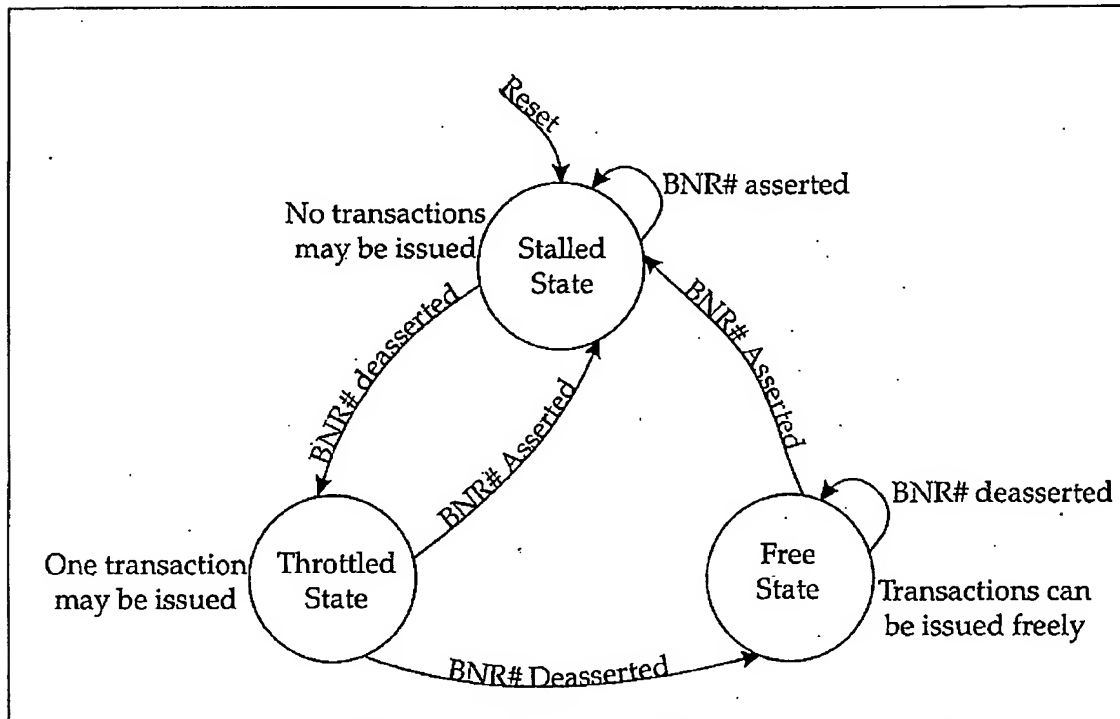
BNR# is a shared, open-drain signal because multiple bus agents may assert it simultaneously to indicate that they are not ready to deal with a new transaction.

Stalled/Throttled/Free Indicator

Each agent capable of initiating transactions must maintain an internal indicator referred to as the stalled/throttled/free indicator. Refer to Figure 10-10 on page 232. At powerup or when reset is asserted, all initiators (i.e., request agents) reset this indicator to the stalled state. All request agents are required to start sampling BNR# on a periodic basis starting soon after reset is removed (see “BNR# Behavior at Powerup” on page 233) and must remain in the stalled state until BNR# is sampled deasserted. This prevents any agent from issuing a transaction until BNR# is sampled deasserted, indicating that all agents are prepared to deal with a new transaction.

Pentium Pro Processor System Architecture

Figure 10-10: Stalled/Throttled/Free Indicator States



Open Gate, Let One Out, Close Gate

When BNR# is sampled deasserted the first time, all request agents transition their indicators from the stalled to the throttled state. This gives permission to the current request signal group owner to initiate a transaction request. If BNR# is asserted on the next sampling as well, all request agents transition the indicator from throttled back to the stalled state. This one time deassertion and then immediate reassertion of BNR# is analogous to opening the gate to let one transaction out and then closing the gate again. This mechanism can be used to permit a new transaction to be issued by the next owner only when everyone is ready to deal with it. A debug tool could use BNR# in this manner to permit issuance of and then track just one transaction from inception to completion.

Open Gate, Leave It Open, Let Them All Out

When everyone is stalled and BNR# is then sampled deasserted one time, they all transition to the throttled state and one transaction may be issued by the next

Chapter 10: Obtaining Bus Ownership

request signal group owner. If BNR# is then again sampled deasserted a second time, all request agents transition the indicator from the throttled to the free state. All request agents are then free to issue new transactions as they acquire ownership of the request signal group. This would permit a new transaction request to be issued to the bus once every three clocks.

Gate Wide Open and then Slammed Shut

When the request agents are operating in the free state and BNR# is then sampled asserted, they all transition from the free to the stalled state and are prevented from issuing any new transactions until the next time BNR# is sampled deasserted. As described earlier, they all then transition from stalled to throttled and one new transaction can be issued. Depending on whether BNR# is sampled deasserted or asserted at the next sample point, they all then transition from throttled to free, or from throttled back to stalled, respectively.

BNR# Behavior at Powerup

Figure 10-11 on page 235 illustrates the behavior of BNR# during and immediately after the assertion of reset.

1. If BNR# was asserted before the assertion of reset, it must be deasserted within one clock after reset is sampled asserted. In addition, all request agents must reset their stalled/throttled/free indicators to the stalled state.
2. The first BNR# sample point is two clocks after reset is sampled deasserted. If it is sampled asserted, all request agents must remain in the stalled state.
3. When asserted, BNR# should remain asserted for only one clock.
4. When request agents are in the stalled state, they sample BNR# every two clocks.
5. On clock 14, BNR# is sampled deasserted and all request agents change to the throttled state one clock later (clock 15). This permits the request agent that is the winner of the first arbitration to issue one transaction request (in clocks 15 and 16).
6. BNR# is sampled every two clocks when in the throttled state. It is sampled deasserted for a second time on clock 16, causing all request agents to transition to the free state one clock later (in clock 17).
7. As the agents transition to the free state, they begin sampling BNR# three clocks after ADS# is asserted by any agent.

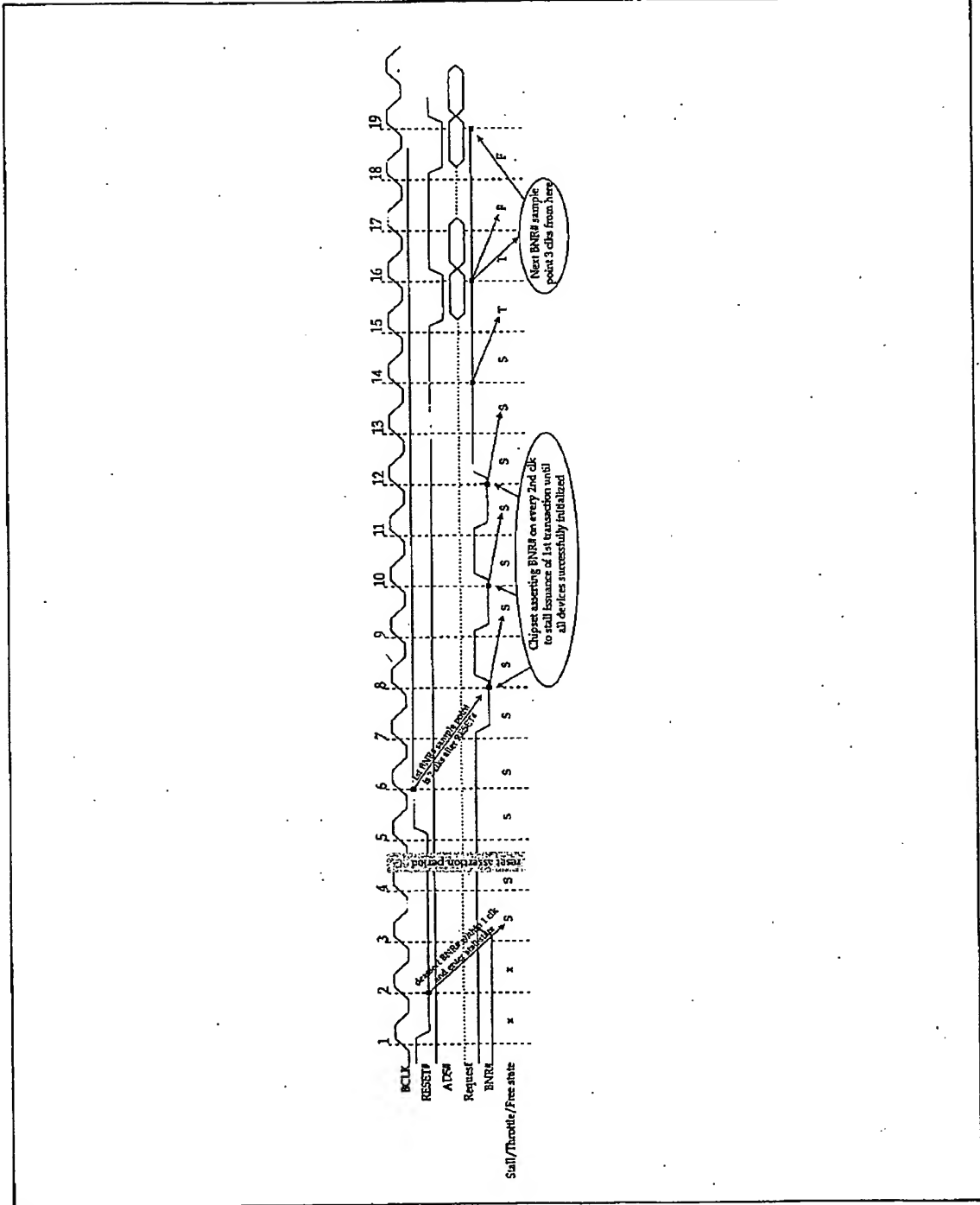
Pentium Pro Processor System Architecture

BNR# and the Built-In Self-Test (BIST)

Any processor that has been instructed to perform its BIST at the trailing-edge of reset will assert BNR# until it has completed its BIST. This is done because the processor's local APIC is incapable of taking part in the automatic selection of the bootstrap processor while its BIST is still in progress. In other words, it cannot receive BIPI messages issued by the local APICs in other processors until its BIST completes. None of the processors will issue BIPI messages until BNR# is sampled deasserted. For information on initiation of a processor's BIST, see "Run BIST Option" on page 39. For information on the selection of the bootstrap processor, see "Selection of Bootstrap Processor (BSP)" on page 55.

Chapter 10: Obtaining Bus Ownership

Figure 10-11: BNR# at Powerup or After Reset



Pentium Pro Processor System Architecture

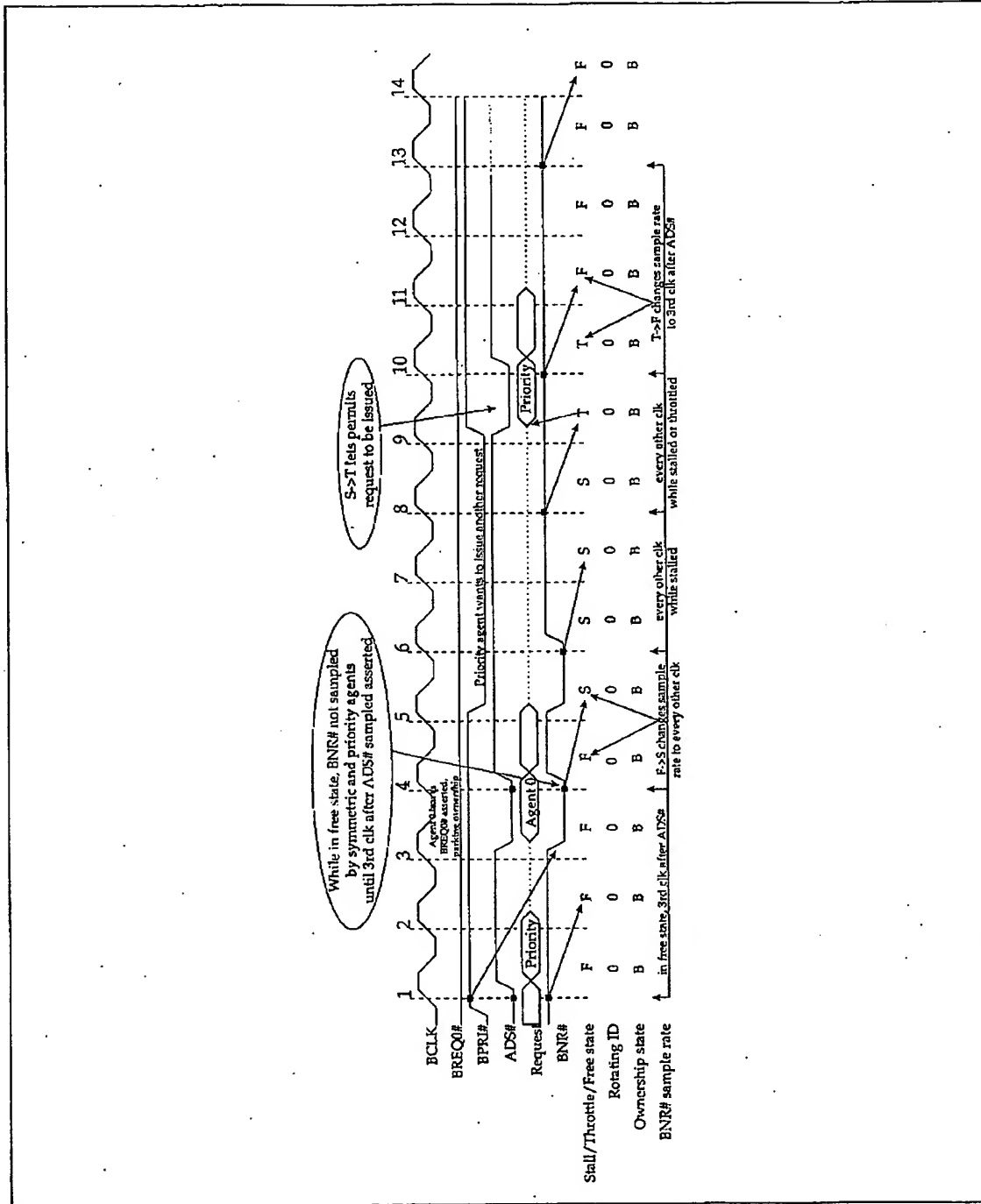
BNR# Behavior During Runtime

Figure 10-12 on page 237 illustrates BNR# behavior after reset (i.e., during runtime).

1. The priority agent had acquired ownership of the request signal group in the clock prior to clock 1 and initiated a transaction request. It deasserted BPRI# as it did so to yield ownership to symmetric agent 0 (BREQ0# was asserted). The indicator is in the free state and BNR# is being sampled by all request agents every three clocks.
2. On clock one, agent 0 samples BPRI# deasserted, indicating that it will be the next owner of the request signal group.
3. Agent 0 initiates a transaction request on clock three (its indicator is in the free state, giving it permission to do so).
4. On clock four, BNR# is sampled asserted, indicating that one or more bus agents cannot handle the issuance of any more transactions. The indicator transitions from the free to the stalled state in clock five and the BNR# sample rate changes to every two clocks. No new transactions can be issued by any request agent while they are in the stalled state.
5. In clock five, the priority agent reasserts BPRI# because it wants to issue another transaction. It cannot do so, however, until BNR# is sampled deasserted.
6. BNR# is sampled deasserted on clock eight and all request agents change the indicator to the throttled state in clock nine. This gives the priority agent permission to issue a new transaction request in clock nine.
7. BNR# is sampled deasserted again on clock 10 and all request agents transition to the free state in clock 11. The BNR# sample rate changes back to every three clocks.

Chapter 10: Obtaining Bus Ownership

Figure 10-12: BNR# During Runtime



Hardware

Section 3:

The Transaction Phases

The Previous Section

The chapters that comprised Part 2, Section 2 introduced the processor's bus and transaction protocol.

This Section

The chapters that comprise Part 2, Section 3 provide a detailed description of each phase that a transaction passes through from inception to completion. It consists of the following chapters:

- "The Request and Error Phases" on page 241.
- "The Snoop Phase" on page 257.
- "The Response and Data Phases" on page 277.

The Next Section

Part 2, Section 4 covers additional transaction and bus topics not covered in the previous sections.

1

11 *The Request and Error Phases*

The Previous Chapter

Before a request agent can issue a new transaction to the bus, it must arbitrate for and win ownership of the request signal group. The previous chapter covered all of the issues related to obtaining request signal group ownership.

This Chapter

This chapter provides a detailed description of the request and error phases of any transaction.

The Next Chapter

The error phase of a transaction is always immediately followed by the snoop phase. The next chapter provides a detailed description of the snoop phase of a transaction.

Caution

As stated earlier in the book, unless noted otherwise the representation of all signal states in tables is in logical, not electrical values. As an example, the first row in Table 11-4 on page 247 shows a 00000b on REQ[4:0]#. indicates a deferred reply transaction type. This means that REQ[4:0]# are deasserted (electrical ones) when driven onto REQ[4:0]#.

Pentium Pro Processor System Architecture

Request Phase

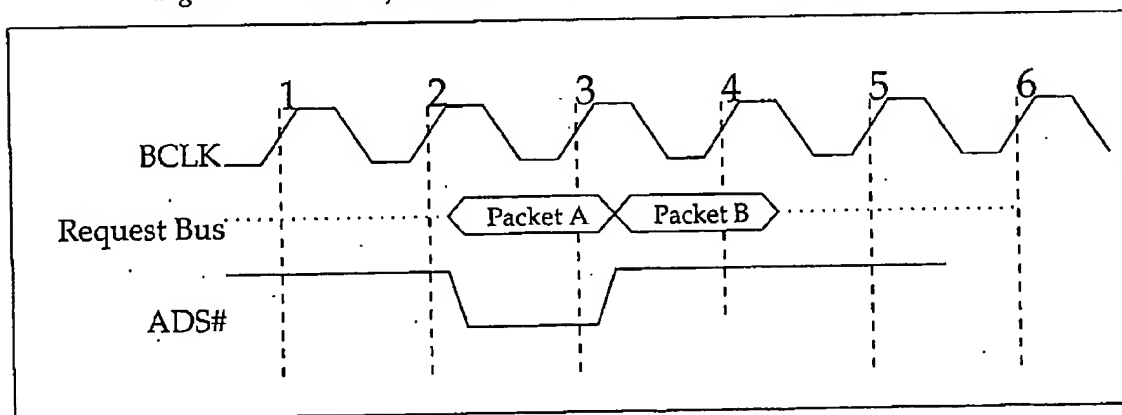
Introduction to the Request Phase

Once ownership of the request signal group has been acquired (see "Obtaining Bus Ownership" on page 201), the request agent uses the request signal group to broadcast the transaction request. This includes the address and transaction type, as well as additional information about the transaction. The request signal group consists of the following signals:

- A[35:3]#. Address bus.
- AP[1:0]#. Address bus parity bits.
- REQ[4:0]#. Request bus.
- RP#. Request bus parity bit.
- ADS#. Address Strobe.

The request phase is always two clocks in duration. The information about the transaction is output in two packets (see Figure 11-1 on page 242), one during each clock. ADS# is asserted during the first clock and deasserted during the second. Its assertion indicates that a new transaction request is being broadcast. All bus agents, not only response agents (i.e., targets), latch both packets. As discussed earlier, all agents must track the transaction as it passes through each phase from inception to completion. In addition, if it is a memory transaction, snoop agents (i.e., processors with internal caches) must submit the memory address to their caches for a lookup and must deliver the snoop result during the transaction's snoop phase. The response agents must decode the address and transaction type to determine which of them is the target of the transaction.

Figure 11-1: Two Information Packets Broadcast during Request Phase



Chapter 11: The Request and Error Phases

Request Signal Group is Multiplexed

There is more transaction-related information to be output during the request phase than there are pins on the processor. To address this problem, the same signal group is used during each of the two clocks that comprise the request phase, but different information is output on these pins during each clock. Intel refers to the two information packets as packets A and B.

In the Intel data book, the information output in the two packets is referred to in the following manner:

- the information output on the address (A[35:3]#) and request (REQ[4:0]#) signal groups during Packet A is referred to as Aa[35:3]# and REQa[4:0]#.
- the information output on the address (A[35:3]#) and request (REQ[4:0]#) signal groups during Packet B is referred to as Ab[35:3]# and REQb[4:0]#.

The information output during Packet B are actually internal signals that are gated onto the address and request pins during the second clock. Table 11-1 on page 243 indicates the names of the signals driven onto A[35:3]# during the second clock.

Table 11-1: Packet B Signal Names

Output Pin(s)	Signal Names
A[31:24]#	ATTR[7:0]#. Attribute signals. See Table 11-8 on page 251.
A[23:16]#	DID[7:0]#. Deferred ID. See Table 11-9 on page 251.
A[15:8]#	BE[7:0]#. The eight byte enable signals indicate which of the bytes in the currently-addressed quadword are to be transferred (i.e., read or written).
A[7:3]#	EXF[4:0]#. Extended function signals. See Table 11-11 on page 252.

Introduction to the Transaction Types

The transaction types currently defined for the bus are listed in Table 11-2 on page 244. The table provides a brief description of each transaction type.

Pentium Pro Processor System Architecture

Table 11-2: Currently-Defined Transaction Types

Transaction Type	Brief Description
Deferred Reply	Initiated by a response agent that had deferred the completion of an earlier transaction (because it was going to take a long time). Addresses the request agent that had initiated the previously-deferred transaction and is used to deliver the transaction completion to the originator. For more information, refer to "Transaction Deferral" on page 307.
Interrupt Acknowledge	Generated by a processor in response to an interrupt from an 8259 interrupt controller to read the interrupt vector. For more information, refer to "Interrupt Acknowledge Transaction" on page 334.
Special Transaction	Generated by the processor to broadcast a message regarding an internal event (e.g., shutdown, halt, etc.). For more information on the Special Transaction, refer to "Central Agent Transactions" on page 333.
Branch Trace Message	Generated by the processor when a branch is taken. Writes out the address of the branch instruction as well as the branch target address. For more information on the Branch Trace Message transaction, refer to "Central Agent Transactions" on page 333.
IO Read	Generated by the processor when executing an IN or INS instruction to read data or status from an IO device. Can also be generated by an agent other than a processor (e.g., a host/PCI bridge).
IO Write	Generated by the processor when executing an OUT or OUTS instruction to write data or a command to an IO device. Can also be generated by an agent other than a processor (e.g., a host/PCI bridge).

Chapter 11: The Request and Error Phases

Table 11-2: Currently-Defined Transaction Types (Continued)

Transaction Type	Brief Description
Memory Read and Invalidate	Generated by the processor to gain exclusive ownership of a cache line. Caused by a cache read miss when performing a read/modify/write (RMW) operation, or before performing a RMW on a line in the cache in the shared state. Also generated when a write miss occurs in a WB memory area. For more information, refer to "Locking—Shared Resource Acquisition" on page 221.
Memory Code Read	Generated by the processor when fetching instructions from memory.
Memory Data Read	Generated by the processor when executing an instruction that requires the reading of data from memory (i.e., a load).
Memory Write (may not be retried)	Generated by the processor when writing back a modified line to memory (a cast out) to make room for a new line in the L1 data cache.
Memory Write (may be retried)	Generated by the processor when executing an instruction that requires the writing of data to memory (i.e., a store).

Contents of Request Packet A

Table 11-3 on page 246 defines the information driven onto the request signal group during the first clock of the request phase. It consists of the address and transaction type. This is sufficient information for response agents to begin the decode to determine which of them is the target device. Table 11-4 on page 247 details the encoding of the transaction types.

Pentium Pro Processor System Architecture

Table 11-3: Request Packet A

Signal(s)	Description
ADS#	Address Strobe is asserted by the agent issuing the request, indicating that the agent is providing all of the request information during this clock and the one that follows. This information represents the address, the request type, and additional information about the transaction.
A[35:3]#	The quadword-aligned IO or memory address is presented to the other agents. If this is a deferred reply transaction, the address of the request agent that initiated the previously-deferred transaction is presented.
REQ[4:0]#	Request. The request (i.e., transaction type) is presented on these signal lines. Table 11-4 on page 247 defines the request codes currently defined for the Pentium Pro processor.
AP[1:0]#	Address Parity bits 1 and 0. AP1# is an even parity bit that covers A[35:24]#, while AP0# is an even parity bit that covers A[23:3]#. Each bit must either be a low or a high to force an even number of electrically low signals on the set of covered signals plus the respective parity signal.
RP#	Request Parity. RP# is an even parity bit that covers REQ[4:0]# and ADS#. RP# must either be a low or a high to force an even number of electrically low signals on the set of covered signals plus the parity signal.

Chapter 11: The Request and Error Phases

Table 11-4: Request Types (note: 0 = signal inactive, 1 = signal active)

Packet A (Request)					Packet B (Extended Request)					Request Type
REQ4#	REQ3#	REQ2#	REQ1#	REQ0#	REQ4#	REQ3#	REQ2#	REQ1#	REQ0#	
0	0	0	0	0	x	x	x	x	x	Deferred Reply. Initiated by response agent to complete a request that was received earlier. For more information, see "Transaction Deferral" on page 307.
0	0	0	0	1						Reserved (ignore).
0	1	0	0	0	DSZ field. Always 00b (= 64-bit data bus width) for Pentium Pro. Ignored by responder.		x	0	0	Interrupt Acknowledge. Generated by a processor when an interrupt request is received from an 8259A interrupt controller.
0	1	0	0	1				1		Special Transaction. For more information, see "Central Agent Transactions" on page 333.
								1	x	Reserved (Central Agent Response).
								0	0	Branch Trace Message. For more information, see "Central Agent Transactions" on page 333.
0	1	0	0	1				1		Reserved (Central Agent Response).
								1	x	Reserved (Central Agent Response).
1	0	0	0	0				LEN		IO Read. For more information, see "IO Transactions" on page 329.
										IO Write. For more information, see "IO Transactions" on page 329.
1	1	0	0	x				x	Reserved (ignore)	
ASZ (Table 11-6 on page 248)		0	1	0				LEN		Memory Read and Invalidate. For more information, see "Locking—Shared Resource Acquisition" on page 221.
				1						Reserved (memory write).
		1	0	0						Memory Code Read (REQ1# in Packet A = 0).
			1							Memory Data Read (REQ1# in Packet A = 1).
			0	1						Memory Write (don't retry). REQ1# in Packet A = 0. Only used when casting modified line to memory to make room for new line in L1 data cache.
				1						Memory Write (may be retried). REQ1# in Packet A = 1.

Pentium Pro Processor System Architecture

Table 11-5: Data Transfer Length Field (see Table 11-4 on page 247)

LEN[1:0]#	State of Byte Enables	Size of Transfer
00b	Specify bytes to be transferred within addressed quadword.	Quadword or subset of quadword.
01b	All asserted.	16 bytes (two quadwords). Pentium Pro doesn't use, but other agents, such as a host/PCI bridge, may.
10b	All asserted.	32 bytes (four quadwords).
11b	Not defined.	Reserved

Table 11-6: Address Space Size Field (see Table 11-4 on page 247)

ASIZ[1:0]#	Description
00b	The memory address is within the range from 0 through 4GB - 1. It should be decoded by any memory targets (i.e., response agents) that reside (at least partially) below the 4GB boundary.
01b	The memory address is within the range from 4GB through 64GB - 1. It should be decoded by any memory targets (i.e., response agents) that reside (at least partially) above the 4GB boundary, but below the 64GB boundary.
10b	Reserved.
11b	Reserved.

32-bit vs. 36-bit Addresses

All of the memory transaction types contain an address size field (see ASZ in Table 11-4 on page 247 and Table 11-6 on page 248). Please note that although the spec only currently defines two ASZ bit patterns, there are two more available for future assignment. In other words, Intel may design future processors with address buses wider (or, although unlikely, more narrow) than the current 36 bits. It is currently permissible to design:

Chapter 11: The Request and Error Phases

1. **32-bit address request agent.** Request agents that are only capable of generating 32-bit memory addresses and only use up to A[31]# (they don't drive A[35:32]#). They are therefore only capable of addressing memory that resides in the lower 4GB. Whenever a request agent of this type generates a memory transaction, it must indicate (on the ASZ lines) that it is only generating a 32-bit address on the lower part of the bus to address a memory target that resides below the 4GB boundary.
2. **32-bit address memory response agent.** Memory response agents that only latch and decode up to A[31]#. By definition, then, these memory response agents reside in the lower 4GB of memory space. A memory agent of this type must check the state of the ASZ lines received in request packet A and only decode the address if ASZ indicates that it is below the 4GB boundary. If it paid no attention to ASZ, it may decode the lower 32 bits of a 36-bit address destined for a memory target that resides above the 4GB boundary. This could result in two memory response agents driving bus signals simultaneously.
3. **36-bit address request agent.** Request agents that are capable of generating 36-bit memory addresses and use up to A[35]#. They are therefore capable of addressing memory that resides in the lower 64GB. Whenever a request agent of this type generates a memory transaction, it must indicate (on the ASZ lines) whether it is generating an address above or below the 4GB boundary.
4. **36-bit address memory response agent.** Memory response agents that latch and decode up to A[35]#. By definition, then, these memory response agents can reside anywhere in the lower 64GB of memory space. A memory agent of this type must check the state of the ASZ lines received in request packet A and only decode the address if ASZ indicates that it is below the 4GB boundary or below the 64GB boundary. If it paid no attention to ASZ, it may decode the lower 36 bits of an address destined for a memory target that resides above the 64GB boundary (this wouldn't happen now, but it could happen in the future). This could result in two memory response agents driving bus signals simultaneously.

Contents of Request Packet B

Table 11-7 on page 250 defines the information driven onto the request signal group during the second clock of the request phase. It contains additional information about the transaction request.

Pentium Pro Processor System Architecture

Table 11-7: Request Packet B

Signal(s)	Description
REQ[4:0]#	Extended Request field (see Table 11-4 on page 247).
A[35:32]#	Optional debug information. Not defined in publicly-released Intel documents.
A[31:24]#	Attribute field. Also referred to as ATTR[7:0]#. Just as it is important for the processor's internal logic to know the rules of conduct within a memory area being accessed, external logic (e.g., an L3 cache) must also behave correctly when a memory access within particular memory region is being accessed. This value is supplied by the MTRR registers and the page table entry for the page being accessed. See Table 11-8 on page 251.
A[23:16]#	Deferred ID field. A response agent may choose to defer completion of a transaction request until a later time. When it is ready to complete the transaction, the response agent must initiate a deferred reply transaction and must address the request agent (using the DID field) that originally initiated the transaction and must also identify which transaction is being completed. Also referred to as DID[7:0]#. See Table 11-9 on page 251.
A[15:8]#	Byte Enables. Also referred to as BE[7:0]#. Specifies the bytes to be transferred within the currently-addressed quadword (see Table 11-5 on page 248). In the case of the special transaction type, the message being broadcast is encoded on the byte enables as indicated in Table 11-10 on page 252.
A[7:3]#	Extended Functions field. Contains additional signals related to SMM, locking, and whether or not the response agent is permitted to defer transaction completion until a later time. Also referred to as EXF[4:0]#. See Table 11-11 on page 252.
AP[1:0]#	Address Parity bits 1 and 0. AP1# is an even parity bit that covers A[35:24]#, while AP0# is an even parity bit that covers A[23:3]#. Each bit must either be a low or a high to force an even number of electrically low signals on the set of covered signals plus the respective parity signal.

Chapter 11: The Request and Error Phases

Table 11-7: Request Packet B (Continued)

Signal(s)	Description
RP#	Request Parity. RP# is an even parity bit that covers REQ[4:0]# and ADS#. RP# must either be a low or a high to force an even number of electrically low signals on the set of covered signals plus the parity signal.

Table 11-8: Attribute Field—Rules of Conduct (see Table 11-7 on page 250)

ATTRIB[7:0]# (A[31:24]#)	Memory Type	Description
00000000b	UC	Address is within memory range designated as uncacheable by MTRRs and/or page table entry.
0000100b	WC	Address is within memory range designated as write-combining by MTRRs and/or page table entry.
00000101b	WT	Address is within memory range designated as write-through by MTRRs and/or page table entry.
00000110b	WP	Address is within memory range designated as write protected by MTRRs and/or page table entry.
00000111b	WB	Address is within memory range designated as write-back by MTRRs and/or page table entry.
all other values	Reserved	

Table 11-9: Deferred ID Composition (see Table 11-7 on page 250)

Bit Field	Description
DID7#	Agent Type. Type 0 = Symmetric agent, while type 1 = priority agent. Delivered by request initiator on A23# during transmission of request Packet B.

Pentium Pro Processor System Architecture

Table 11-9: Deferred ID Composition (see Table 11-7 on page 250) (Continued)

Bit Field	Description
DID[6:4]#	Agent ID of request initiator. Delivered by request initiator on A[22:20]# during transmission of request Packet B.
DID[3:0]#	Transaction ID. Delivered by request initiator on A[19:16]# during transmission of request packet B. Every deferrable transaction (DEN# asserted in packet B on A4#) will have a unique transaction ID. When one of these transactions has passed its snoop phase without being deferred, its deferred ID may be reused by the request agent. When a response agent initiates the deferred reply transaction, it uses the deferred ID as the address of the reply.

Table 11-10: Messages Broadcast Using Special Transaction (see Table 11-7 on page 250)

Message Type	Byte Enable Settings
Shutdown. For more information about any of the messages, see "Central Agent Transactions" on page 333.	00000001b
Flush	00000010b
Halt	00000011b
Sync	00000100b
Flush Acknowledge	00000101b
Stop Grant Acknowledge	00000110b
SMI Acknowledge	00000111b
Reserved	all other encodings

Table 11-11: Extended Function Field (see Table 11-7 on page 250)

Bit	Description
EXF4#	SMMEM#. When asserted, System Management memory is being accessed (rather than regular memory).

Chapter 11: The Request and Error Phases

Table 11-11: Extended Function Field (see Table 11-7 on page 250) (Continued)

Bit	Description
EXF3#	SPLCK#. Split Lock. When asserted, processor is accessing a memory semaphore that splits across cache lines in WB memory or across quadwords in UC or WT memory. Results in locked transaction series consisting of 2 reads to read semaphore from memory followed later by 2 writes to update semaphore. For more information, refer to "Locking—Shared Resource Acquisition" on page 221.
EXF2#	Reserved.
EXF1#	DEN#. Defer Enable. When asserted, response agent is permitted to defer completion until a later time. For more information, see "Transaction Deferral" on page 307.
EXF0#	Reserved.

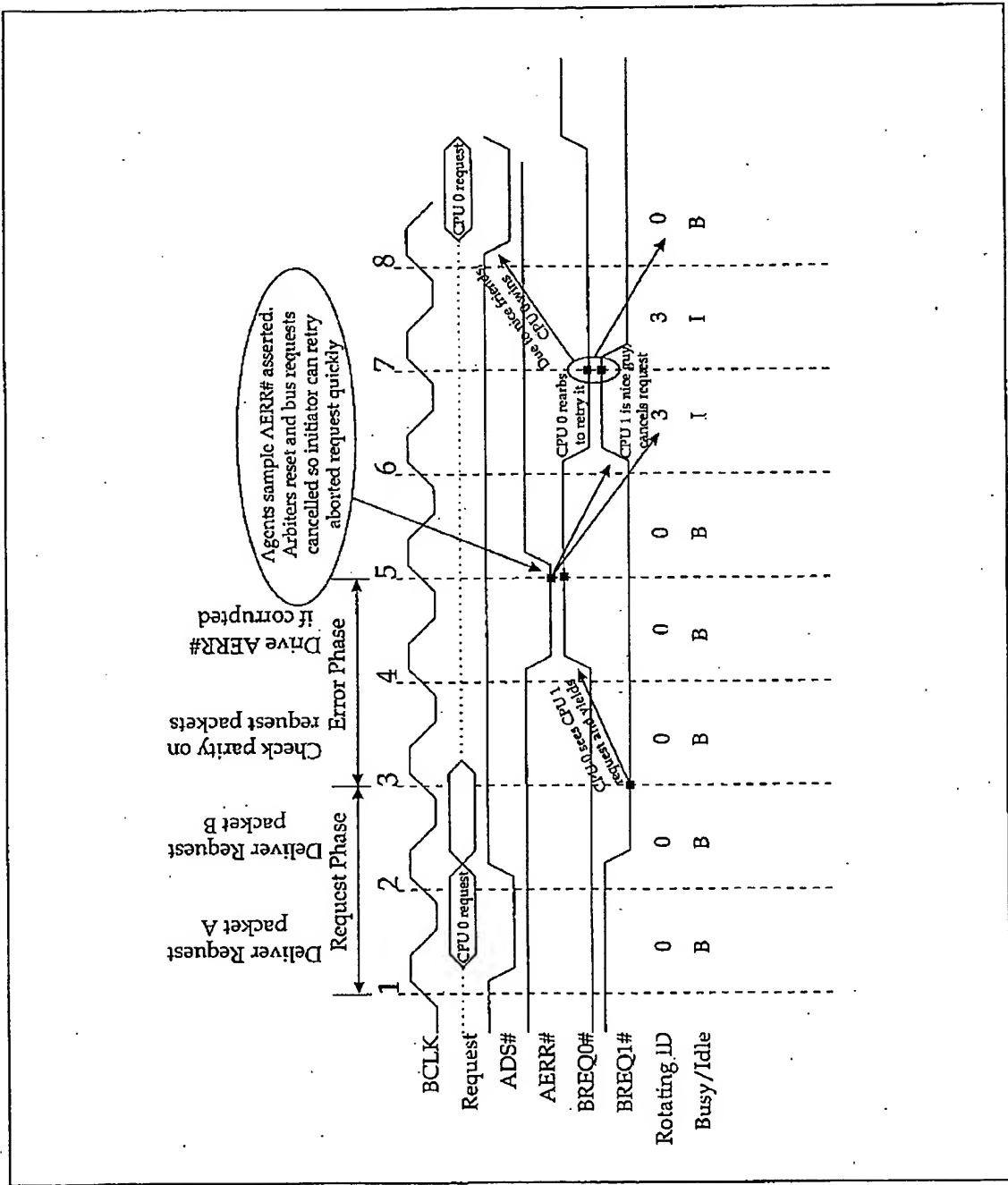
Error Phase

In-Flight Corruption

Refer to Figure 11-2 on page 254. All agents latch request packets A and B of the transaction request on clocks two and three, respectively. The request phase consists of clocks one and two, while the error phase consists of clocks three and four (it is always two clocks in duration). When packet A is latched on clock two, all agents check the parity on AP[1:0]# and RP# during clock two. When packet B is latched on clock three, all agents check the parity for the packet during clock three. If an error is detected in either packet, all agents that received a corrupted packet assert AERR# for one clock during clock four (do not assert AERR# during clock three if packet A is corrupted). Note that AERR# is one of the six GTL+ signals that is a shared, open-drain signal that can be driven by multiple devices simultaneously.

Pentium Pro Processor System Architecture

Figure 11-2: Error Phase



Chapter 11: The Request and Error Phases

Who Samples AERR#?

Request Agent

If it was enabled to do so at powerup time (see “Error Observation Options” on page 39), the request agent that initiated the transaction request samples AERR# on clock five. If AERR# is sampled deasserted, the transaction request was received correctly by all other bus agents and the transaction may proceed. If AERR# is sampled asserted, however, the transaction is cancelled (i.e., it is deleted from the request agent’s IOQ).

Other Bus Agents

All other bus agents latch packets A and B of the transaction request. If they were enabled to do so at powerup time (see “Error Observation Options” on page 39), they sample AERR# on clock five. If AERR# is sampled deasserted, the transaction remains in their IOQs and will be tracked until its completion. If AERR# is sampled asserted, however, the transaction is cancelled (i.e., it is deleted from the agents’ IOQs).

Who Drives AERR#?

Any bus agent that was enabled to drive AERR# at powerup time (see “Program-Accessible Startup Features” on page 45), will assert AERR# in clock four if it detects that either of the transaction request packets were corrupted in flight.

Request Agent’s Response to AERR# Assertion

If the request agent samples AERR# asserted on clock five, it cancels (i.e., aborts) the transaction. Optionally, it may reattempt the transaction. Whether it retries and the number of retry attempts is device-specific. *The Pentium Pro processor has a retry count of one for all transactions that result in AERR#.* If AERR# is sampled asserted and the transaction will be retried, the request agent asks for request signal group ownership again in clock six. If it’s a symmetric agent and it had deasserted its BREQn# when it initiated the transaction, it reasserts its BREQn# signal. If it’s a priority agent and it had deasserted BPRI# when it initiated the transaction, it reasserts BPRI#.

Pentium Pro Processor System Architecture

If the retry also fails, the Pentium Pro processor generates a machine check exception (if CR4[MCE] = 1) and latches the address and transaction type into the machine check architecture registers (see "Machine Check Architecture" on page 415).

Other Guys are Very Polite

When they sample AERR# asserted, any other request agents that might have been requesting ownership of the request signal group deassert their requests for one clock. This opens a window for the request agent that experienced the bad request phase to win ownership to reattempt the transaction (on clock eight). However, all of the other symmetric agents had already sampled the BREQn# signals and registered which of them wanted the bus next (normally, symmetric agents are not permitted to deassert an already asserted BREQn# line until has gained ownership; however, this is a special case to help out a friend who has fallen on hard times). To avoid confusing their arbitration mechanisms, all of the symmetric agents reset their rotating ID to three and the ownership state to idle.

12

The Snoop Phase

The Previous Chapter

The previous chapter provided a detailed description of the request and error phases of a transaction.

This Chapter

Unless the transaction is cancelled by an error in its error phase, the error phase of a transaction is always immediately followed by the snoop phase. This chapter provides a detailed description of the snoop phase.

The Next Chapter

The snoop phase of the transaction is always immediately followed by the response and possibly the data phase. The next chapter provides a detailed description of the response and data phases.

Agents Involved in Snoop Phase

Refer to Figure 12-1 on page 259. The following agents are involved in the snoop phase of the transaction:

- The **request agent** issues the transaction request. This can be one of the processors, or one of the host/PCI bridges. If the transaction is a memory transaction, it also checks the snoop result presented in the snoop phase.
- The **snoop agents** are the processors. They latch the transaction and, if it's a memory transaction, submit the memory address to their internal caches for a lookup. They present the snoop result to the request agent. If the snoop resulted in a hit on a modified line in a processor's cache, that snoop agent (i.e., processor) supplies the modified line in the data phase (see next bullet). If it's a non-memory transaction, the processors do not snoop the transaction.

Pentium Pro Processor System Architecture

- The response agent is the currently-addressed target. This could be main memory, the configuration registers or IO ports within the memory controller or within one of the host/PCI bridges, a target residing on one of the PCI buses, or a target residing on the ISA or EISA bus. If the response agent is main memory, it must observe the snoop response presented by the snoop agents. If the access results in a miss on all the caches, or in a hit on a clean line, the memory controller supplies the read data or, if a write, accepts the write data presented by the request agent. However, if the transaction is a read that hits on a modified line, the snoop agent supplies the modified line directly to the request agent and also writes it to memory (in other words, the transaction started out a read from main memory's perspective and turned into a write, but it remained a read from the request agent's perspective). If the transaction is a write that hits on a modified line, the memory controller accepts the write data from the request agent, then accepts the modified line from the snoop agent, and finally merges the write data into the modified line and writes the resulting line into memory.

Pentium Pro Processor System Architecture

Snoop Phase Has Two Purposes

In the snoop phase, the request agent samples the snoop result signals to determine:

1. if the currently-addressed response agent (i.e., the target) intends to complete the transaction now or it will be retried or its completion deferred until a later time (because it will take a while to complete).
2. if the transaction is a memory read or write that the response agent will complete now (i.e., it doesn't intend to defer its completion), does any other cache have a copy of the line and, if so, in what state (clean or modified) will its line be at the completion of the transaction?

Snoop Result Signals are Shared, DEFER# Isn't

There are three snoop result signals (divided into two groups) that are sampled by the request agent during the snoop phase:

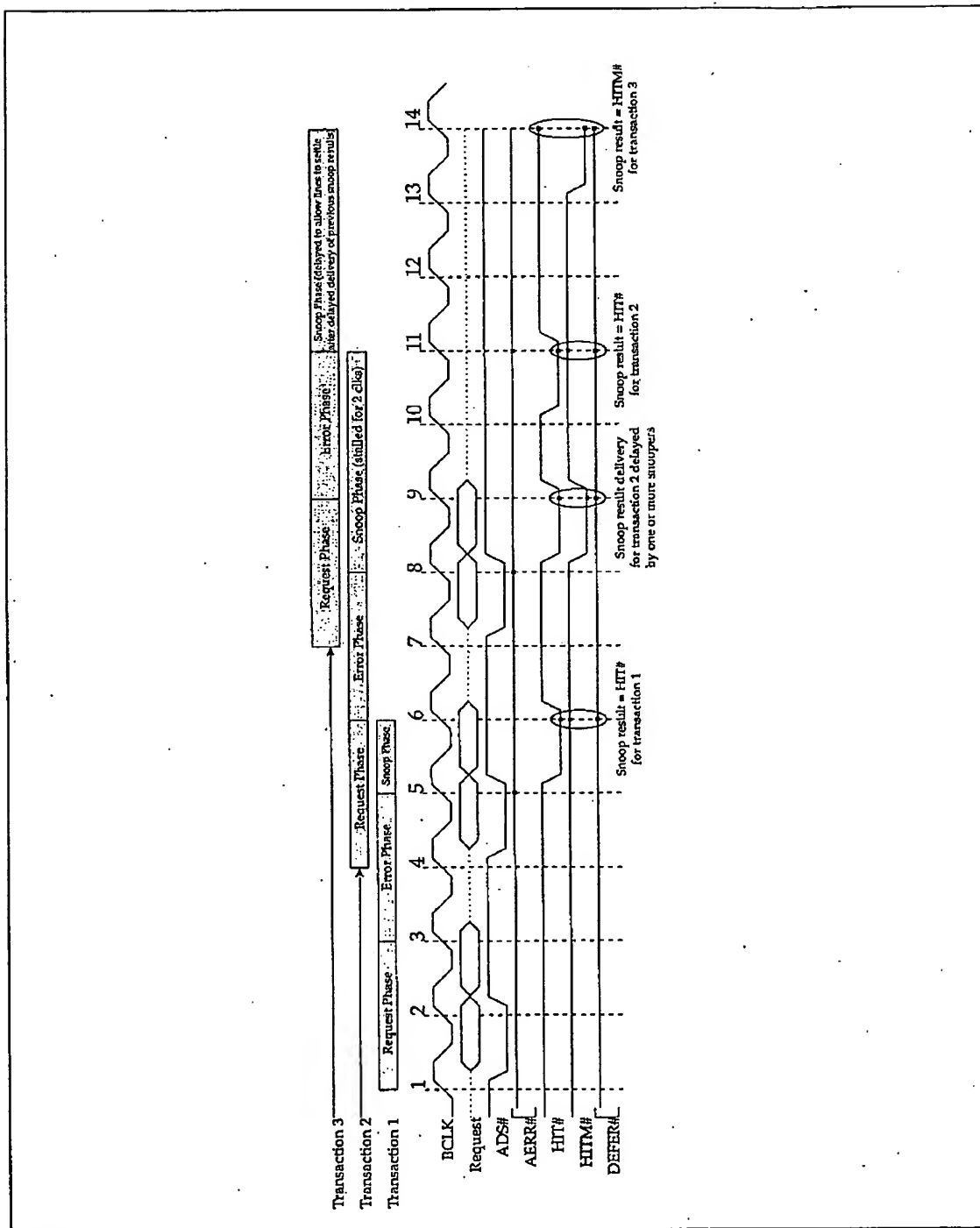
1. The HIT# and HITM# are the signals used by the snoop agents (i.e., the caches) to deliver the cache snoop result, or to stall the completion of the snoop phase if one or more of the snoop agents isn't ready to deliver the snoop result (see "Line in E or M State" on page 228 for an example of snoop stall). Both HIT# and HITM# might be driven by multiple snoopers if they need to stall the completion of the snoop phase until they have the snoop result available. At that time, each of the snoopers would stop driving both lines and either drive neither (if it's a miss), just HIT# (if it's a hit on an E or S line), or just HITM# (if it's a hit on a modified line). HIT# and HITM# are shared, open-drain signals that may be driven by more than one device at a time.
2. Only the currently-addressed response agent (i.e., the target) is permitted to assert the DEFER# signal during the snoop phase, so it is not a shared, open-drain signal. The response agent will only assert DEFER# if it intends to issue a retry or a deferred response to the request agent. These topics are discussed towards the end of this chapter.

Snoop Phase Duration Is Variable

Refer to Figure 12-2 on page 261. The snoop phase begins immediately after the error phase completes (clocks 5, 8, and 11) and completes when a valid snoop result (something other than both HIT# and HITM# asserted) is presented to the request and response agents by the snoop agents. Table 12-1 on page 262 defines the meaning of the various snoop results.

Chapter 12: The Snoop Phase

Figure 12-2: Snoop Phase



Pentium Pro Processor System Architecture

1. In Figure 12-2 on page 261, transaction one's snoop phase completes one clock after it starts (on clock six) when HIT# is sampled asserted with HITM# and DEFER# deasserted. This indicates a hit on a line in the E state in one cache, or a hit on lines in the S state in two or more processors.
2. Transaction two's snoop phase starts on clock eight and the snoop result is first sampled one clock later on clock nine. However, in this case, both HIT# and HITM# are sampled asserted, indicating a snoop stall (one or more of the snoop agents aren't ready to deliver the actual snoop result). As a result, the request agent stretches the duration of the snoop phase and then samples the snoop result again on clock 11. This time a good result is sampled—HIT# is asserted alone, indicating a hit on a line in the E state in one cache, or a hit on lines in the S state in two or more processors.
3. Transaction three's snoop phase starts on clock 11 when its error phase ends. Because the snoop result for the previous transaction was delayed, transaction three's request agent delays its first sampling of the current transaction's snoop result until three clocks after the previous transaction's snoop result was sampled. This allows two clocks for the lines to settle and one for the current transaction's snoop result to be driven. After the snoop agents deliver the previous transaction's snoop result, they turn off their drivers to allow HIT# or HITM# to return high. In order to let these shared lines settle, they must delay driving the next snoop result until two clocks after they turn their drivers off.

Table 12-1: Snoop Result Table (0 = deasserted, 1 = asserted)

HIT#	HITM#	DEFER#	Interpretation
0	0	0	Clean snoop (HIT# and HITM# both deasserted indicates a miss in all caches) and transaction completion not deferred by response agent.
0	0	1	Transaction completion deferred by response agent. Ignore the snoop result. In the response phase, the response agent will issue either a retry or a deferred response. If it issues a deferred response, the real snoop result will be delivered later in the deferred reply transaction initiated by the response agent. If it issues a retry response, the request agent must retry the transaction from scratch later.
0	1	0	Hit on a modified line. Snoop agent with modified line will supply the line to the response agent in the data phase (and to the request agent if it's a read).
0	1	1	Hit on a modified line. Assertion of HITM# in an unlocked transaction overrides the assertion of DEFER# (in other words, transaction completion will not be deferred). Snoop agent with modified line will supply the line to the response agent in the data phase (and to the request agent if it's a read).

Chapter 12: The Snoop Phase

Table 12-1: Snoop Result Table (0 = deasserted, 1 = asserted) (Continued)

HIT#	HITM#	DEFER#	Interpretation
1	0	0	Hit on a clean line in one or more caches.
1	0	1	At least one cache has a copy of the line in the clean state and the response agent's assertion of DEFER# indicates that it will defer completion of the transaction until a later time. The request agent ignores assertion of HIT#. In the response phase, the response agent will either issue a retry or a deferred response. If it issues a retry, the request agent must retry the transaction from scratch again later. If the response agent issues a deferred response, when the response agent initiates the deferred reply at a later time, the response agent will deliver the snoop result to the request agent. The request agent uses this deferred snoop result to choose the state of its copy of the line.
1	1	0	HIT# and HITM# both asserted indicates that at least one snoop agent cannot deliver the snoop result yet. This results in a snoop stall. The request agent waits two clocks before sampling the snoop result again. The stall continues until a good snoop result is delivered. Note that the state of DEFER# is ignored by the request agent until a good snoop result is delivered (something other than 11b on HIT# and HITM#).
1	1	1	Same as previous entry.

Is There a Snoop Stall Duration Limit?

No. The spec doesn't define a limit on how long the snoop phase may be stretched. However, to avoid hanging the system in the event of a failure, it is advisable for the chipset to monitor for excessive wait states inserted into the snoop phase and signal an error (e.g., via BERR#) if this condition occurs.

Memory Transaction Snooping

Snoop's Effects on Caches

When a memory transaction is snooped, the snoop result affects both the request agent and the snoop agents. There are a number of possible cases. The type of memory transaction, whether or not other caches have a copy of the line, and the state of the line(s) in other caches all have an effect. In addition, sometimes the length of the data transfer has an effect. Table 12-2 on page 264 defines the scenarios and the results.

Pentium Pro Processor System Architecture

Table 12-2: Effects of Snoop

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory read	Miss on all caches	Data read from memory. If a code cache line fill, the line is placed in the code cache in the S state (the code cache is SI, not MESI) and in the E state in the L2 cache. If a data cache line fill, the line is placed in the L2 and data caches in the E state.
	Hit on E or S line in one or more caches	Data read from memory. If a code cache line fill, the line is placed in the code and L2 caches in the S state. If a data cache line fill, the line is placed in the L2 and data caches in the S state. If a snoop agent had a copy of the line in the E state, it changes it to the S state.
	Hit on M line in one cache	The operation started as a read from memory, but this changes it to a read of the M line from the snoop agent's cache. From the memory controller's standpoint, the read has turned into an implicit writeback of the M line to memory. The memory controller issues an implicit writeback response to the request agent in the response phase and accepts the line from the snoop agent during the data phase. The snoop agent changes its copy from the M to the S state. The request agent may have requested less than a line, but a full line is returned in toggle mode order, critical quadword first. The request agent takes what it requested and ignores the rest. If it requested the whole line, the line is placed in the L2 and code or data caches in the S state.

Chapter 12: The Snoop Phase

Table 12-2: Effects of Snoop (Continued)

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory write	Miss on all caches	Data written to memory. No effect on caches.
	Hit on E or S line in one or more caches	Snoopers indicate a miss. Invalidates lines in other caches (because the processors don't have the ability to snarf data being written to memory by other agents).
	Hit on M line in one cache	<p>Case 1: writing less than a line. Memory controller accepts write data from request agent, then accepts implicit writeback of full line from snoop agent's cache. Memory controller then merges write data into writeback line and writes resultant line into memory. Snoop agent invalidates its copy of the line.</p> <p>Case 2: writing a full line. Memory controller accepts write data from request agent, then asserts TRDY# to snoop agent to indicate its readiness to accept the implicit writeback of full line from snoop agent's cache. The snoop agent can respond in one of two ways:</p> <ul style="list-style-type: none"> • If it doesn't check the length of the write data, it asserts DBSY# and uses the data bus to supply the modified line to the memory controller. The memory controller then merges write data into writeback line and writes resultant line into memory. Alternately, the memory controller could discard the modified line. • Alternatively, it could note that the request agent is writing a full line and decide that it won't send the modified line to the response agent (i.e., the memory controller). This is indicated by not asserting DBSY# in response to the assertion of TRDY#. The memory controller writes the line received from the request agent into memory. <p>In either case, the snoop agent invalidates its copy of the line.</p>

Pentium Pro Processor System Architecture

Table 12-2: Effects of Snoop (Continued)

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory read and invalidate for 32 bytes	Miss on all caches	Data is read from memory and is placed in the data cache. The transaction is basically a read with intent to modify. If the programmer immediately modified one or more locations within the line (e.g., update of a semaphore), the line is now in the M state. If the programmer chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on E or S line in one or more caches	Snoopers indicate a miss. Data is read from memory and is placed in the data cache. The line(s) in the other caches are invalidated. The transaction is basically a read with intent to modify. If the programmer immediately modified one or more locations within the line (e.g., update of a semaphore), the line is now in the M state. If the programmer chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on M line in one cache	Data is supplied to the request agent and the response agent (i.e., the memory controller) from the modified line in the snoop agent's cache. Snoop agent invalidates its copy of the line. Response agent (i.e., the memory controller) may or may not write the line into memory. Since the line was read with the intent to modify it once it is placed in the data cache, the memory controller designer may choose not to waste memory bandwidth. If the programmer immediately modified one or more locations within the line (e.g., update of a semaphore), the line is now in the M state. If the programmer chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.

Chapter 12: The Snoop Phase

Table 12-2: Effects of Snoop (Continued)

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory read and invalidate for 0 bytes	Miss on all caches	Processor would not perform this transaction because it already had a copy of the line in its data cache (why else would it be reading 0 bytes?) in the E state (E because no one else had a copy). If it didn't have a copy itself, it would be performing a read and invalidate for a full line, not 0 bytes.
	Hit on E or S line in one or more caches	<ul style="list-style-type: none"> In the case where another processor has a copy in the E state, the processor performing the transaction would have had a miss on its cache and would be performing a read and invalidate for a full line, not 0 bytes. In the case where the processor performing the transaction has a copy in the S state, the transaction kills the line in the other caches that have it in the S state. The snoopers indicate a miss. If the programmer immediately modified one or more locations within the line (e.g., update of a semaphore), the line is now in the M state. If the programmer chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on M line in one cache	If another processor has a copy of the line in the M state, then the processor performing the transaction doesn't have a copy. It would therefore have a miss on its data cache and would perform a read and invalidate for a full line, not 0 bytes.

After Snoop Stall, How Soon Can Next Snoop Result be Presented?

Because the snoop result for the previous transaction was delayed, the next transaction's request agent delays its first sampling of the current transaction's snoop result until three clocks after the previous transaction's snoop result was sampled. This allows two clocks for the lines to settle and then one for the current transaction's snoop result to be driven. After the snoop agents deliver the previous transaction's snoop result, they turn off their drivers to allow HIT# or HITM# to return high. In order to let these shared lines settle, they must delay driving the next snoop result until two clocks after they turn their drivers off.

Pentium Pro Processor System Architecture

Self-Snooping

For an example of self-snooping, refer to "Self-Modifying Code and Self-Snooping" on page 173.

Non-Memory Transactions Have a Snoop Phase

All transactions, including non-memory transactions, have a snoop phase. For non-memory transactions, however, there are only three valid snoop responses:

1. Clean snoop (HIT# and HITM# both deasserted).
2. Defer (DEFER# asserted), indicating that the currently-addressed target will issue a retry or a deferred response in the transaction's response phase.
3. Snoop stall (HIT# and HITM# both asserted), indicating that the snoop phase is to be extended by two more clocks. If necessary, a non-memory response agent is permitted to stall the snoop response to give itself more time for internal operations to complete before presenting clean snoop or defer as the snoop result.

The non-memory transaction types are the deferred reply, interrupt acknowledge, special, branch trace message, IO read, and IO write transactions. For more information on these transaction types, refer to "Central Agent Transactions" on page 333, "IO Transactions" on page 329, and "Transaction Deferral" on page 307.

Transaction Retry and Deferral

Permission to Defer Transaction Completion

The request agent uses the state of its DEN# output (Defer Enable) in packet B of the request phase (see "Contents of Request Packet B" on page 249) to grant or deny the response agent permission to defer the transaction's completion until a later time. Note that this does not deny the response agent permission to retry the transaction.

- When DEN# is deasserted, permission to defer is denied. Permission is denied under the following circumstances:
 - when a processor is performing a memory write to cast a modified line back to memory to make room for a new line being read into the data cache.

Chapter 12: The Snoop Phase

- when an agent is performing a locked transaction series.
- when an agent is performing a deferred reply transaction.
- An agent that requires fast completion of its transaction may deassert DEN#, denying the response agent permission to defer the transaction for later completion.
- When DEN# is asserted, permission is granted to assert DEFER# and issue a deferred response.

DEFER# Assertion Delays Transaction Completion

When the currently-addressed response agent does not assert DEFER# in the snoop phase, the agent is promising to complete the transaction now (with something other than a retry or a deferral for later completion).

If DEFER# is asserted in the snoop phase, the response agent will either issue a retry or a deferred response to the request agent in the response phase. In both of these cases, the response agent is saying that it cannot complete the transaction now. There are two possible cases:

1. If the request agent must complete this transaction before it can proceed with subsequent transactions (in other words, it's important for proper device driver operation that its transactions complete in program-initiated order), then the request agent must not issue any new transactions until it successfully completes the current transaction at a later time (by reattempting the retried transaction or receiving the deferred reply transaction from the response agent).
2. If the request agent doesn't have to complete this transaction before issuing subsequent transactions, it can choose to issue different transactions before reattempting the retried transaction or receiving the deferred reply transaction from the response agent.

Transaction Retry

If a target cannot deal with the current transaction right now (e.g., due to a temporary logic busy condition), it may choose to assert DEFER# in the snoop phase and issue a retry response even if DEN# was deasserted in packet B of the request phase.

When the request agent samples DEFER# asserted in the snoop phase followed by a retry response in the response phase, it is obligated to retry the transaction from scratch on a periodic basis until it completes the transaction. All agents,

Pentium Pro Processor System Architecture

including the request agent, delete the transaction from their IOQ buffers when the retry response is seen in the response phase. If the transaction involves a data transfer, the data phase is cancelled.

Transaction Deferral

Mail Delivery Analogy

A good analogy for a deferred transaction is sending a letter with a request for delivery notification. The read or write transaction is received and suspended until, at a later time, the response agent sends a delivery notice back to the transaction originator (and, if it's a read, delivers the requested data). Note that the response agent may have had a problem when delivering the transaction and must then report the problem faithfully.

Example System Operation Overview

Refer to the system model pictured in Figure 12-3 on page 271 during the following discussion. The illustration shows a Pentium Pro processor cluster of four processors. The processors share access to memory.

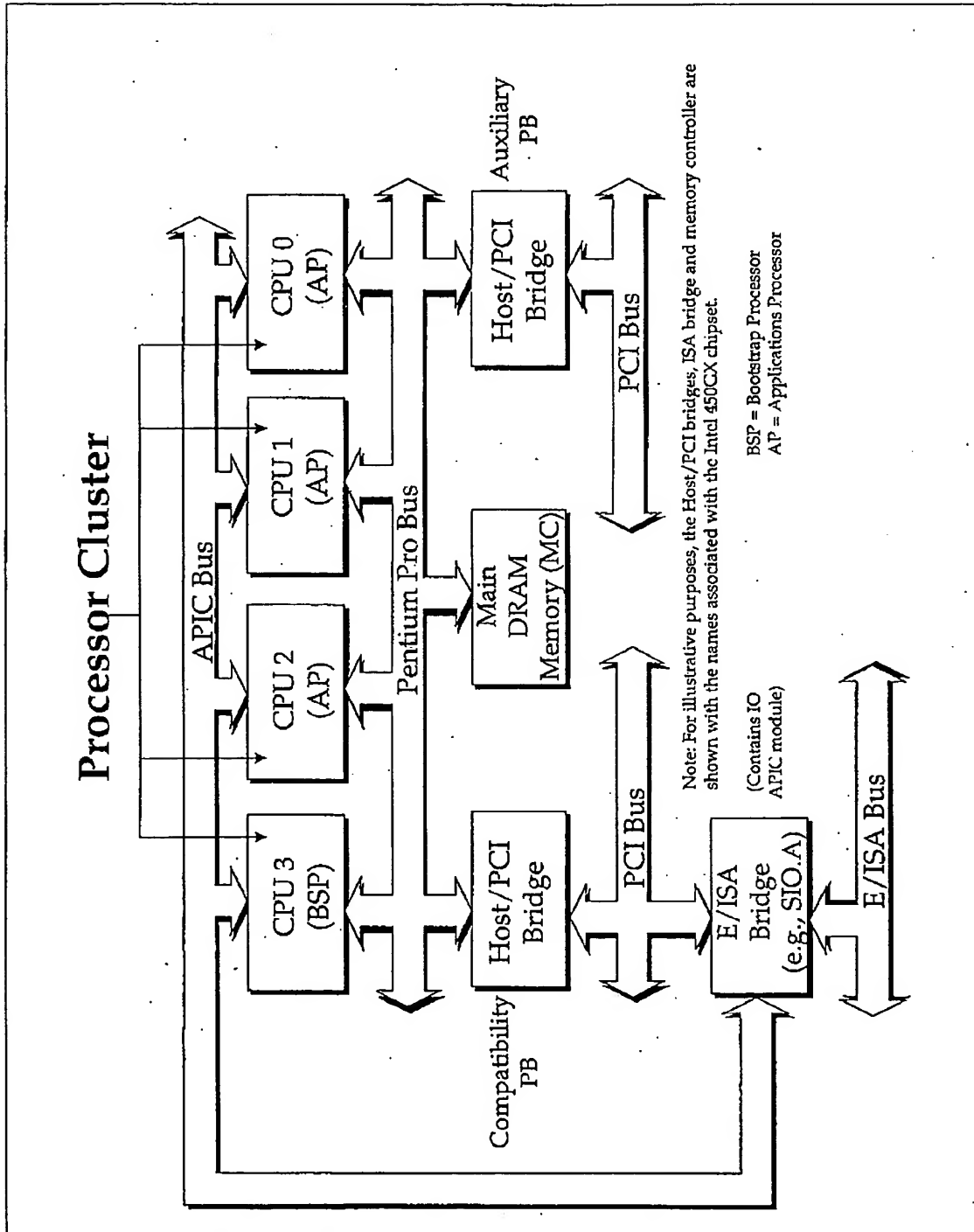
Any processor may issue a transaction request targeting a device residing beyond either one of the host/PCI bridges. During the startup process, the host/PCI bridges are configured to recognize the IO and memory ranges assigned to the PCI and EISA or ISA devices that reside beyond the bridge.

When any of the processors attempts to perform a read or write with a target residing on either of the PCI buses or on the EISA or ISA bus, it can result in very long latency during the data phase of the transaction.

The Wrong Way. When the transaction is initiated, all of the agents on the processor bus latch the transaction and each of the response agents examines the address and transaction type to determine which of them is the targeted response agent. Assuming that the processor is targeting a device that resides beyond one of the host/PCI bridges, that bridge must act as the response agent for the transaction. Essentially, it is the surrogate for the addressed target which resides somewhere on the other side of the bridge. If the transaction is a read, the bridge takes ownership of the processor data bus (by asserting DBSY#), but keeps DRDY# deasserted until it has the requested read data. It then presents the data to the processor and asserts DRDY# to indicate its presence.

Chapter 12: The Snoop Phase

Figure 12-3: System with Four Processors and Two Host/PCI Bridges



Pentium Pro Processor System Architecture

Before the bridge can provide the requested read data from the target or can deliver the write data to the target, it must first arbitrate for ownership of the PCI bus. When it asserts its REQ# to the PCI bus arbiter, the following conditions may be true:

- the PCI bus is currently in use by a PCI master performing a burst read or write transaction with another device on the PCI bus.
- one or more other bus masters that reside on the PCI bus may also be asserting their REQ# to the PCI bus arbiter.
- the host/PCI bridge may have been the last owner of the PCI bus and the arbiter may use a rotational priority scheme.

If all of the above conditions are true, the host/PCI bridge will not attain ownership of the PCI bus until each of the other PCI masters has each received ownership of the PCI bus. Upon attaining ownership, each master may then retain ownership of the bus until it has exhausted its assigned timeslice (i.e., master latency timer, or MLT). When the host/PCI bridge finally attains ownership (that could be quite a while) and initiates the read or write PCI transaction, it must then wait up to five PCI clocks for the PCI target to claim the transaction (i.e., assert DEVSEL#). Once the target has claimed the transaction, it will issue a retry to the bridge if the transaction will take more than 16 clocks from its start (the assertion of FRAME#) to transfer (read or write) the first dword. This 16 clock rule was added in the 2.1 PCI specification. The target has memorized the transaction and is processing it off-line. The host bridge must then periodically retry the transaction until the target finally hands over the read data or indicates that it has accepted the write data. The absolute worst-case scenario occurs if the targeted device resides on the ISA bus (an extremely slow, 8MHz bus populated by horrendously slow devices).

If the host/bridge had kept DRDY# deasserted on the processor bus during this entire period of time, the data bus would remain busy (DBSY# asserted). This means that all transaction requests (that require a data transfer) subsequently issued by any bus agents (including the same one) will stall.

The result—gridlock!

The Right Way. Rather than tie up the processor bus, the bridge asserts DEFER# to the processor in the snoop phase and issues the deferred response in its response phase. This informs the processor that the transaction will be completed at a later time. The processor moves the transaction from its IOQ to its deferred transaction queue. All other bus agents delete the transaction from their IOQs. The bridge has “memorized” the following information:

Chapter 12: The Snoop Phase

- Whether or not the response agent is permitted to defer the transaction (refer to “Permission to Defer Transaction Completion” on page 268).
- The memory address and transaction type (refer to “Contents of Request Packet A” on page 245).
- The DID field delivered in packet B of the request phase (refer to “Contents of Request Packet B” on page 249). This will permit the bridge to “address” the processor later when it initiates its deferred reply transaction to complete the original transaction.
- If the transaction is a memory read, the amount of data to be read (“Contents of Request Packet B” on page 249).
- If the transaction is a write, the data to be written.

When the read or write transaction completes on the PCI bus, the bridge must deliver completion notification to the processor that initiated the read or write. To do this, it arbitrates for processor bus ownership (by asserting BPRI#). When it has acquired ownership of the request signal group, it initiates a deferred reply transaction. The DID that was originally received from the processor is used as the address in request packet A. All bus agents latch the transaction request and detect that it is a completion notice for a previously-issued transaction that is still awaiting completion. The processor that originally initiated the transaction has a match on its agent ID and on the transaction ID, identifying this as the completion notice for the previously-deferred transaction.

Bridge Should be a Faithful Messenger

During the snoop phase of the deferred reply transaction, the bridge and the processor change roles—the processor becomes the request agent and the bridge becomes the snoop agent. *The deferred reply transaction is never snooped by snoop agents.* Rather, the bridge now supplies the snoop result that it received on the PCI bus to the processor (since there aren’t any caches on PCI buses, the result is a miss—HIT# and HITM# are deasserted).

In the response phase, the bridge acts as the response agent and delivers the response it received from the PCI target:

- If the transaction was a write and the data was delivered to the PCI target successfully, the response is the No Data response.
- If the transaction was a write and the data wasn’t delivered to the PCI target successfully (due to master abort, target abort, or a parity error), the response is the Hard Failure response and the transaction failed.
- If the transaction was a read and the data was read from the PCI target successfully, the response is the Normal Data response and the read data is supplied in the data phase.

Pentium Pro Processor System Architecture

- If the transaction was a read and the data wasn't read successfully, the response is the Hard Failure response and the transaction failed.

Detailed Deferred Transaction Description

For a more detailed description of deferred transactions, refer to "Transaction Deferral" on page 307.

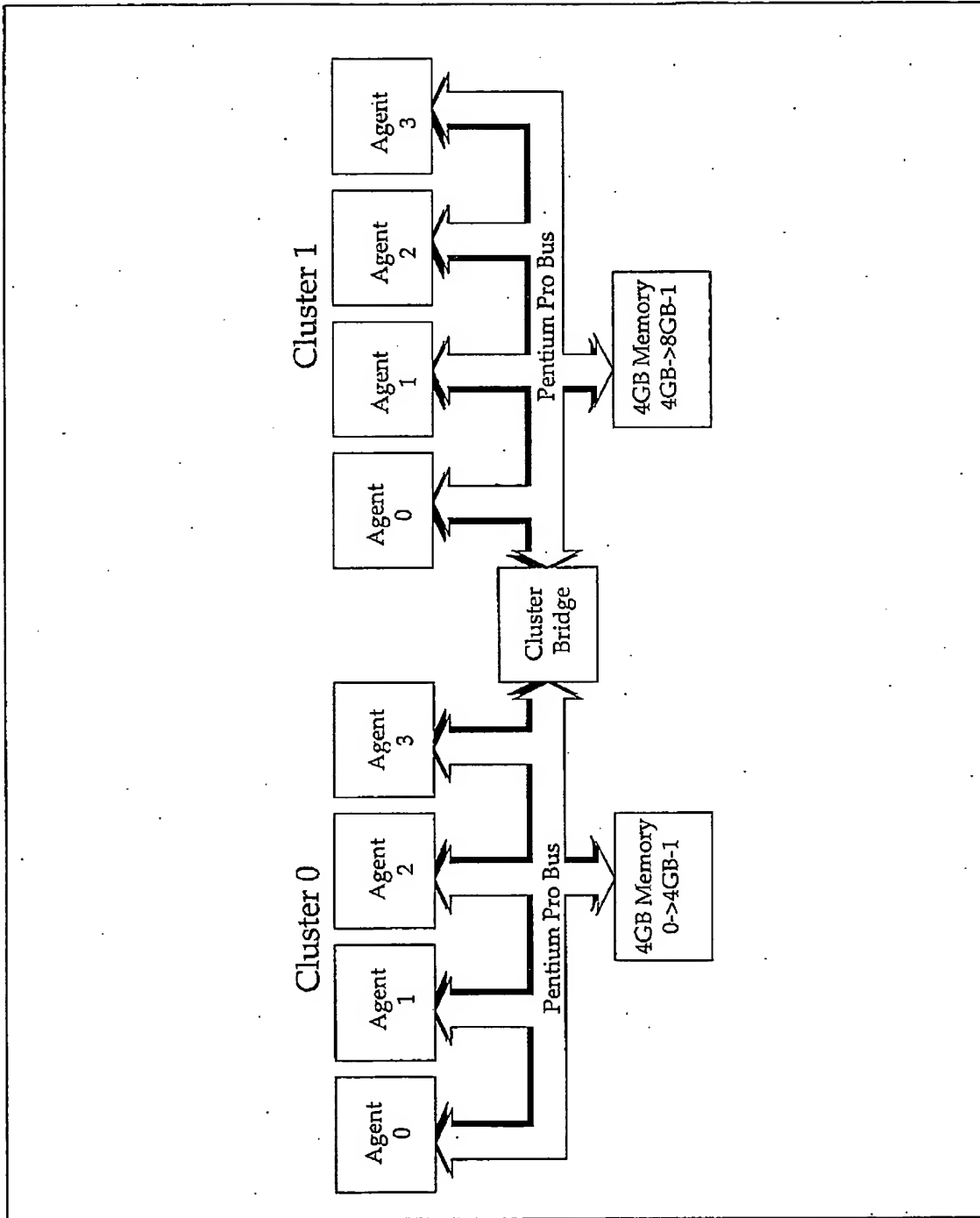
What if HITM# and DEFER# both Asserted?

For all unlocked transactions (i.e., LOCK# not asserted), assertion of HITM# overrides the assertion of DEFER#. In other words, if HITM# is asserted, the response agent provides the implicit writeback response in the response phase and the snoop agent writes the modified line to memory (and, if it's a read transaction, the line is also received by the request agent).

Consider a system with two processor clusters linked by a cluster bridge (see Figure 12-4 on page 275). A bus 0 processor initiates a memory read from the bus 1 memory controller that hits on a modified line in the cache of one of the bus 0 processors. In other words, the bus 0 processor with the modified copy of the line had read the line from the bus 1 memory controller earlier in time and had subsequently made one or more changes to the line in its data cache. In this case, the most up-to-date copy of the line is the modified one in the cache. Rather than passing the read to the bus 1 memory controller, the bus 0 cache therefore supplies the read data to the request agent and to the cluster bridge. The cluster bridge arbitrates for ownership of bus 1 and writes the fresh copy of the line to the bus 1 memory controller.

Chapter 12: The Snoop Phase

Figure 12-4: Two Processor Clusters Linked by Cluster Bridge



Pentium Pro Processor System Architecture

How Does Locking Change Things?

If DEFER# and HITM# are both asserted in the snoop phase of the first of a locked transaction series (typically a memory read to read a semaphore, segment descriptor, page table entry, etc.), the transaction is not deferred. Rather, the snoop agent supplies the modified line to memory (and to the request agent) as an implicit writeback operation.

When the transaction completes, the request agent is required to re-arbitrate and must repeat the transaction and assert LOCK# again. The Intel data book does not explain the rationale for this and as of this writing, the author has been unable to come up with it on his own (very frustrating).

It is forbidden for the response agent to respond to the second and any subsequent transactions of the locked series with DEFER# without HITM# also asserted. The data book says that this is considered to be a protocol violation and goes on to say that the response agent must issue a retry response in the response phase, forcing the request agent to retry the entire locked transaction series from the beginning. The author finds this confusing. The data book makes it sound like the response agent, which violated the spec by issuing DEFER# in the snoop phase of something other than the first transaction of the locked series, is then responsible for punishing itself and the request initiator by issuing a retry response in the response phase. Why would the response agent assert DEFER# knowing full-well that it was violating the spec?

13 *The Response and Data Phases*

The Previous Chapter

The previous chapter provided a detailed description of the snoop phase.

This Chapter

The response phase immediately follows the snoop phase. The snoop result helps the response agent determine what its response to the transaction will be. It should be noted that the response and data phases of a transaction may overlap and that the data phase may also overlap other phases of a transaction. This chapter provides a detailed description of the response and data phases.

The Next Chapter

A response agent may defer completion of a transaction. The next chapter provides a detailed description of deferred transaction handling.

Note on Deferred Transactions

Please note that a detailed description of deferred transactions can be found in the chapter entitled "Transaction Deferral" on page 307.

Purpose of Response Phase

In the response phase of the transaction, the response agent must indicate to the request agent whether:

- it will service the request immediately.
- it can't service it now, but will be able to later.

Pentium Pro Processor System Architecture

- it will service it off-line and get back to the request agent with the completion later.
- it is broken and can't service the request at all.
- a snoop agent has a copy of the targeted line in the modified state and will supply the line to the memory controller.

The possible responses are:

1. the response agent may command the request agent to retry the transaction repeatedly until it succeeds (or fails). In other words, it can't service the request now, but will be able to later.
2. the response agent may inform the request agent that it will defer completion of the transaction until a later time. In other words, it will service the request off-line and get back to the request agent with the completion later.
3. the response agent may indicate a hard failure to the request agent. In other words, it is broken and can't service the request at all.
4. If the transaction is one that doesn't require the response agent to send data to the request agent (i.e., it is a write transaction, a special transaction, or a memory read or memory read and invalidate transaction for 0 bytes), the response agent indicates that, as requested, no data will be returned to the request agent. The request will be service immediately.
5. If the transaction is a memory read or write that results in a hit on a modified line in the snoop phase, the response agent indicates that the snoop agent will transfer the entire modified line to memory (referred to as an implicit writeback operation) in the data phase of the transaction (and, if it's a read transaction, to the request agent at the same time).
6. If the transaction is any form of a read transaction (memory read, memory read and invalidate for 32 bytes, IO read, or interrupt acknowledge), the response agent indicates whether or not (it may choose to defer delivery of the read data until a later time) it will return the requested data in the data phase. This is referred to as the normal data response.

Response Phase Signal Group

The following signals are used in the response phase:

- **RS[2:0]#.** Response bus. Used to deliver the response to the request agent.
- **RSP#.** Response bus parity bit. The parity signal that covers RS[2:0]#. It is an even parity signal that is driven low or high to force an even number of electrical lows in the overall 4-bit pattern that includes RSP#.
- **TRDY# (Target Ready).** Only asserted by the response agent if data is to be written to the response agent by either the request agent, a snoop agent (implicit writeback of a modified line), or both. The assertion of TRDY# indicates the response agent's readiness to accept the write data.

Chapter 13: The Response and Data Phases

Response Phase Start Point

The response phase starts immediately after the snoop phase completes.

Response Phase End Point

The response phase ends when the response agent delivers a valid response to the request agent. This implies that the response agent can stall the response phase (i.e., insert wait states) until it is ready to present its response.

The specification doesn't place a limit on the number of wait states that may be inserted into the snoop phase of a transaction. However, the system designer may choose to monitor the behavior of agents to ensure that none of them inserts excessive wait states. This would adversely affect all subsequently-issued transactions that are awaiting delivery of their snoop result.

List of Responses

Table 13-1 on page 279 lists the possible responses that can be presented on RS[2:0]#.

Table 13-1: Response List (0 = deasserted, 1 asserted)

RS[2:0]#	Description
000b	Idle. This is referred to as the idle state. None of the response signals are asserted. This is the state of RS[2:0]# before and after the response has been delivered to the request agent. In other words, immediately upon entry into the response phase, RS[2:0]# are in this state and will remain in this state until a valid response is presented. When any of the valid responses are driven (for one clock), one or more of the RS[2:0]# signals are driven low. A look at the other table entries shows that all of the valid response patterns have at least one of RS signals asserted (remember that a 1 = asserted, or electrical low). After one clock, the response is removed. The RS signals then all return to the deasserted state (in other words, back to the idle state).
001b	Retry. The response agent may command the request agent to retry the transaction repeatedly until it succeeds (or fails). In other words, it can't service the request now, but will be able to later.

Pentium Pro Processor System Architecture

Table 13-1: Response List (0 = deasserted, 1 asserted) (Continued)

RS[2:0]#	Description
010b	Deferred. The response agent may inform the request agent that it will defer completion of the transaction until a later time. In other words, it will service the request off-line and get back to the request agent with the completion later.
011b	Reserved.
100b	Hard Failure. The response agent may indicate a hard failure to the request agent. In other words, it is broken and can't service the request at all.
101b	No Data. This response indicates that no data was requested by the request agent and no data will therefore be delivered. This is the proper response to a write (although data is written to the device, none is requested from it). It is also the proper response to a transaction that doesn't require any data to be transferred—the special transaction, the memory read and invalidate for 0 bytes, the memory read for 0 bytes, or the IO read for 0 bytes.
110b	Implicit Writeback. This is the response given if the memory transaction resulted in a hit on a modified line (i.e., HITM# was asserted in the snoop phase). This means that the snoop agent that has the modified line will supply the modified line to the response agent (i.e., the memory controller) as well as to the request agent (if it's a read transaction). The author thinks of this as the "don't be startled" response. The request agent may be attempting to read less than a line of information and, if the snoop agent supplies the data, it always sends the full line. The implicit write-back response tells the request agent that four quadwords (32 bytes) will be transferred rather than the smaller data packet actually requested. The four quadwords are transferred in toggle mode order, critical quadword first. This means that the first quadword sent back by the snoop agent will be the first one requested by the request agent and, if a second quadword was also requested (i.e., a 16 byte read request), the second quadword sent back will be the second one expected. The request agent should just take the quadword(s) requested and ignore the rest. The memory controller (i.e., the response agent), on the other hand, will accept the full line.

Chapter 13: The Response and Data Phases

Table 13-1: Response List (0 = deasserted, 1 asserted) (Continued)

RS[2:0]#	Description
111b	Normal Data. This is the proper response to any read request (that doesn't hit on a modified line and is not deferred)—a memory read, a memory read and invalidate for 32 bytes, an IO read, an interrupt acknowledge, or a deferred reply that is returning previously-requested read data.

Response Phase May Complete Transaction

In all transactions that do not involve a data transfer, the delivery of the response ends the transaction. This would include the following transactions:

- the special transaction broadcasts a processor message encoded on the byte enables, but does not read or write data.
- a memory read or memory read and invalidate for 0 bytes transfers no data.
- a read transaction that receives a deferred response from the response agent. In other words, the response agent will initiate a deferred reply transaction at a later time and transfer the read data to the request agent at that time.

Any transaction that performs a read or write that does not receive a deferred, retry, or hard fail response will have both a response and a data phase.

Data Phase Signal Group

Table 13-2 on page 281 defines the signals used during the data phase of a transaction.

Table 13-2: Data Phase-related Signals

Signal(s)	Description
DBSY#	Asserted by agent that will drive data on the data bus (response agent on read; request agent on a write; snoop agent on a snoop hit on modified line) when it takes ownership of the data bus. Stays asserted at least until 1 clk before final transfer completes.

Pentium Pro Processor System Architecture

Table 13-2: Data Phase-related Signals

Signal(s)	Description
DRDY#	Asserted by agent driving data on data bus to indicate data is valid. Can be deasserted during the data phase to insert wait states into the data phase.
D[63:0]#	8 data paths used to transfer up to a quadword at a time.
DEP[7:0]#	Data bus ECC/Parity protection bits. When used as ECC bits, permits correction of single-bit failures and detection of double-bit failures. Each of these bits is related to one of the eight data paths.

Five Example Scenarios

Transaction that Doesn't Transfer Data

Figure 13-1 on page 284 illustrates a transaction that doesn't require a data transfer at all. This would include:

1. Special transaction.
2. Transaction that receives a retry response.
3. Transactions that receive a hard failure response.
4. Read transaction that receives a deferred response (data will be transferred at a later time in another transaction). In addition, the read doesn't hit on a modified line (the case where it hits a modified line is described in "Read that Hits a Modified Line" on page 291).
5. Memory read or memory read and invalidate for 0 bytes.

In these cases, the data bus is not used by the request agent (because it's not a write), the response agent (because no data is being read), or any of the snoopers (because it doesn't hit a modified line).

The response phase is entered immediately upon delivery of the transaction's snoop result (see clock 6). However, the target of this transaction must ensure that it doesn't drive its response until both of the following conditions have been met:

1. The snoop phase for this transaction must have completed. A response agent must check the snoop result (see clock 6) before it can determine what its response will be.

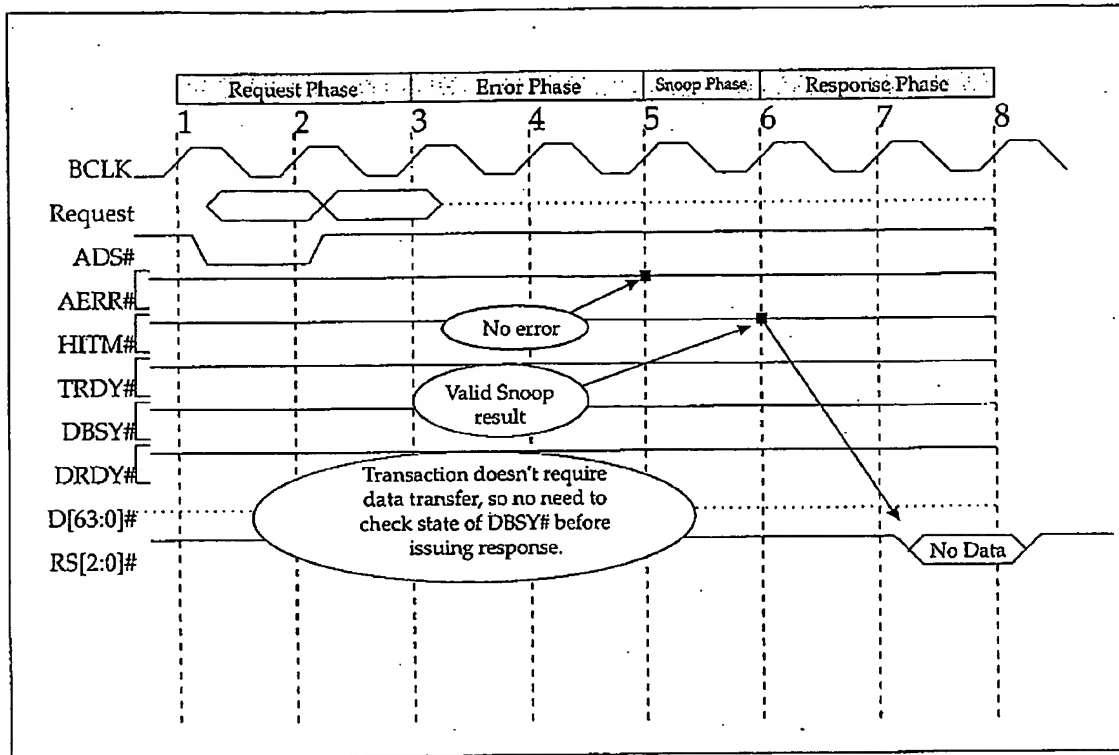
Chapter 13: The Response and Data Phases

2. The response agent observed the target of the previous transaction deliver its response and then idle the response signals. In order to allow signal settling time, it must not drive its response until at least two clocks after the response from the previous response agent is seen. This allows one clock for the previous target to backoff its drivers from RS[2:0]# and one clock for the lines to settle before the target of the current transaction starts to turn on its RS drivers. Remember that it is permissible for the target of a transaction to delay the delivery of its response (by keeping RS[2:0]# deasserted) until is ready to deliver it. It then drives its response (see Table 13-1 on page 279) for one clock and releases the RS[2:0]# signals (which return to the idle state). Assume that the target of the current transaction isn't paying attention (i.e., it's not properly tracking all currently outstanding transactions) and drives its response to its request agent before the target of the previously-issued transaction drives its response to the request agent that addressed it. The request agent that issued the earlier transaction would receive the response from the wrong target for a transaction that it didn't issue. This would well and truly confuse everyone involved. Another possibility is that both targets would simultaneously drive the RS signals, resulting in a wire-ORed (i.e., bogus) result.

The appropriate response for a transaction that doesn't require a data transfer is the no data response (see clock 8).

Pentium Pro Processor System Architecture

Figure 13-1: Transaction That Doesn't Require Data Transfer



Read that Doesn't Hit a Modified Line and is Not Deferred

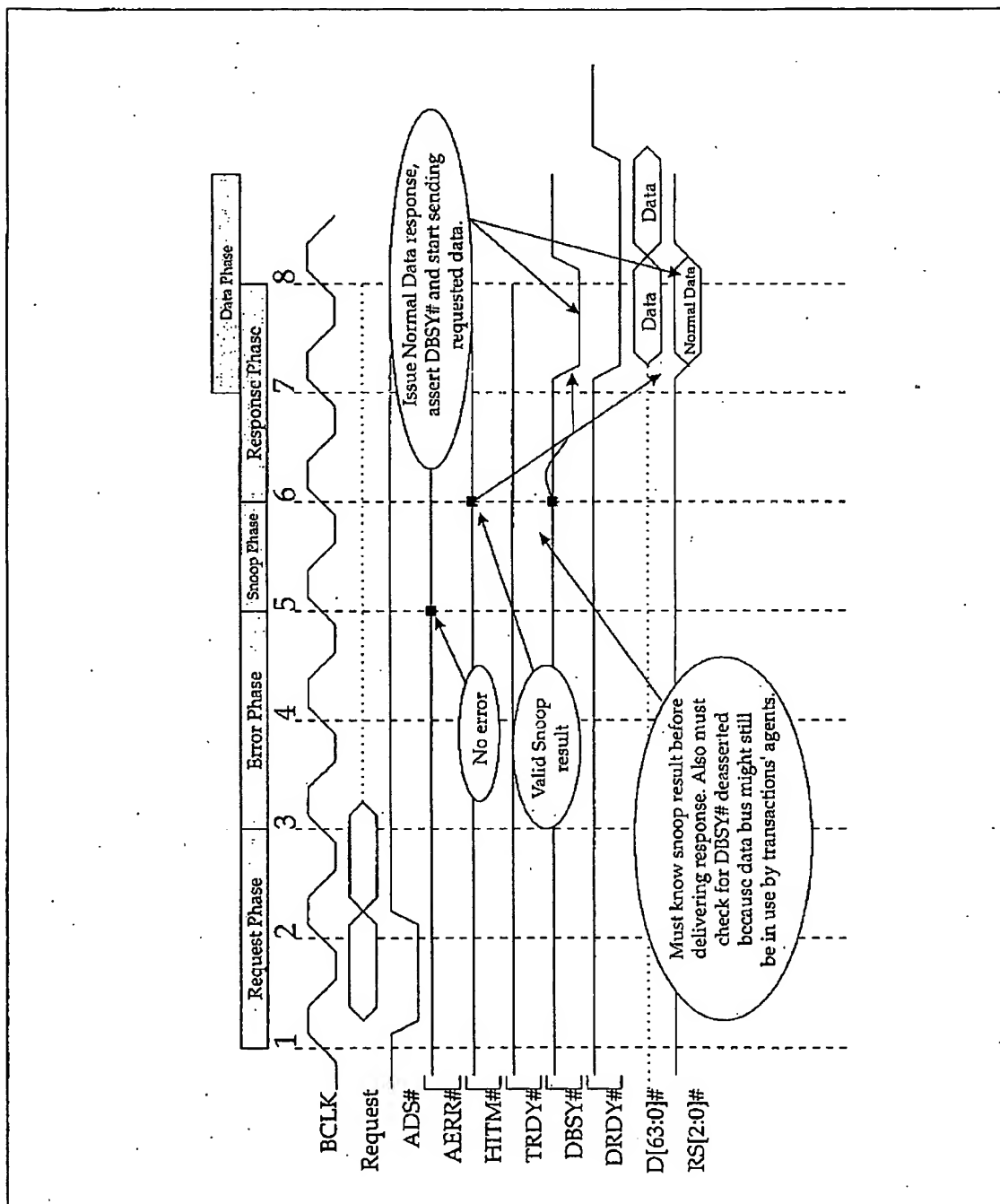
Basics

Figure 13-2 on page 285 illustrates a read for two quadwords (16 bytes) that doesn't hit a modified line and isn't deferred. In other words:

1. the response agent will take ownership of the data bus and transfer the requested data to the request agent.
2. the response agent will supply the Normal Data response at the same time that it takes ownership of the data bus.
3. none of the snoop agents has a copy of the line in the modified state and therefore a snooper will not be writing data to memory.

Chapter 13: The Response and Data Phases

Figure 13-2: Read that Doesn't Hit a Modified Line and Isn't Deferred



Pentium Pro Processor System Architecture

Detailed Description

On a read transaction, the response agent delivers the Normal Data response (see clock 8) if it intends to transfer the requested data. At the same time that it drives the response, it takes ownership of the data bus in preparation for driving the requested read data back to the request agent. Before delivering its response, the response agent must be sure that all of the following conditions are met:

1. The response agent doesn't deliver its response until it has received its snoop result (see clock 6—it needs the snoop result to determine what its response will be).
2. The response agent doesn't deliver its response until the previously-issued transaction's response agent has issued its response and the RS lines have settled (you don't want to confuse the previous transaction's request agent by giving it the response to the next transaction and you don't want to start driving the RS lines until they have quit ringing). At the earliest, the response for the current transaction can be driven three clocks after the previous response was driven—this allows one clock for the previous response to be driven, one clock for the response agent to backoff its drivers, and one clock for the RS lines to settle.
3. If the previously-issued transaction requires a data transfer, the response agent doesn't deliver its response until the request or response agent (whichever is using the data bus) has completed its data transfer and given up ownership of the data bus (i.e., it has deasserted DBSY#; see clock 6).

In Figure 13-2 on page 285, the response agent takes ownership of the data bus by asserting DBSY# in clock 7. When it's ready to drive the first data item (up to eight bytes of data), it asserts DRDY# and drives the data. If it's not ready to drive the data, it asserts DBSY#, but holds off on the assertion of DRDY# until the clock where it begins driving the data.

In the figure, the response agent asserts DBSY# and DRDY# and drives the first quadword onto the data bus in clock 7. On clock 8, the request agent samples the data and the state of the DRDY# signal. DRDY# is asserted, qualifying the data as valid. In clock 8, the response agent leaves DRDY# asserted and drives the second quadword onto the data bus. In addition, it deasserts DBSY#, relinquishing ownership of the data bus. On clock 9, the request agent samples the second quadword from the data bus and the state of DRDY#. DRDY# is asserted, qualifying the second quadword as valid.

Chapter 13: The Response and Data Phases

How Does Response Agent Know Transfer Length?

The request agent delivered the transfer length in the LEN field of packet B during the request phase (refer to "Contents of Request Packet B" on page 249). The possible lengths are:

- One quadword or a subset thereof (the byte enables in packet B indicate which bytes within the addressed quadword).
- Two full quadwords (16 bytes).
- Four full quadwords (32 bytes).

What's the Earliest that DBSY# Can be Deasserted?

The earliest point at which DBSY# can be deasserted is in the clock that the data bus owner is ready to transfer the final data item (clock 8 in Figure 13-2 on page 285). There's no danger of the next data bus owner taking ownership too soon because the next owner won't sample DBSY# deasserted until the next clock and cannot take ownership until one clock after that. This provides the current data bus owner with one clock to drive the final data item and one clock to backoff from the data bus before the next owner can assert DBSY# and take ownership of the data bus.

Relaxed DBSY# Deassertion

The specification permits what is referred to as relaxed DBSY# deassertion. In clock 8 of Figure 13-2 on page 285, DBSY# is deasserted in the clock that the data bus owner is ready to transfer the final data item. It is permissible to keep DBSY# asserted until the final data item is transferred (on clock 9), and then deassert both DRDY# and DBSY# at the same time.

Write that Doesn't Hit a Modified Line and Isn't Deferred

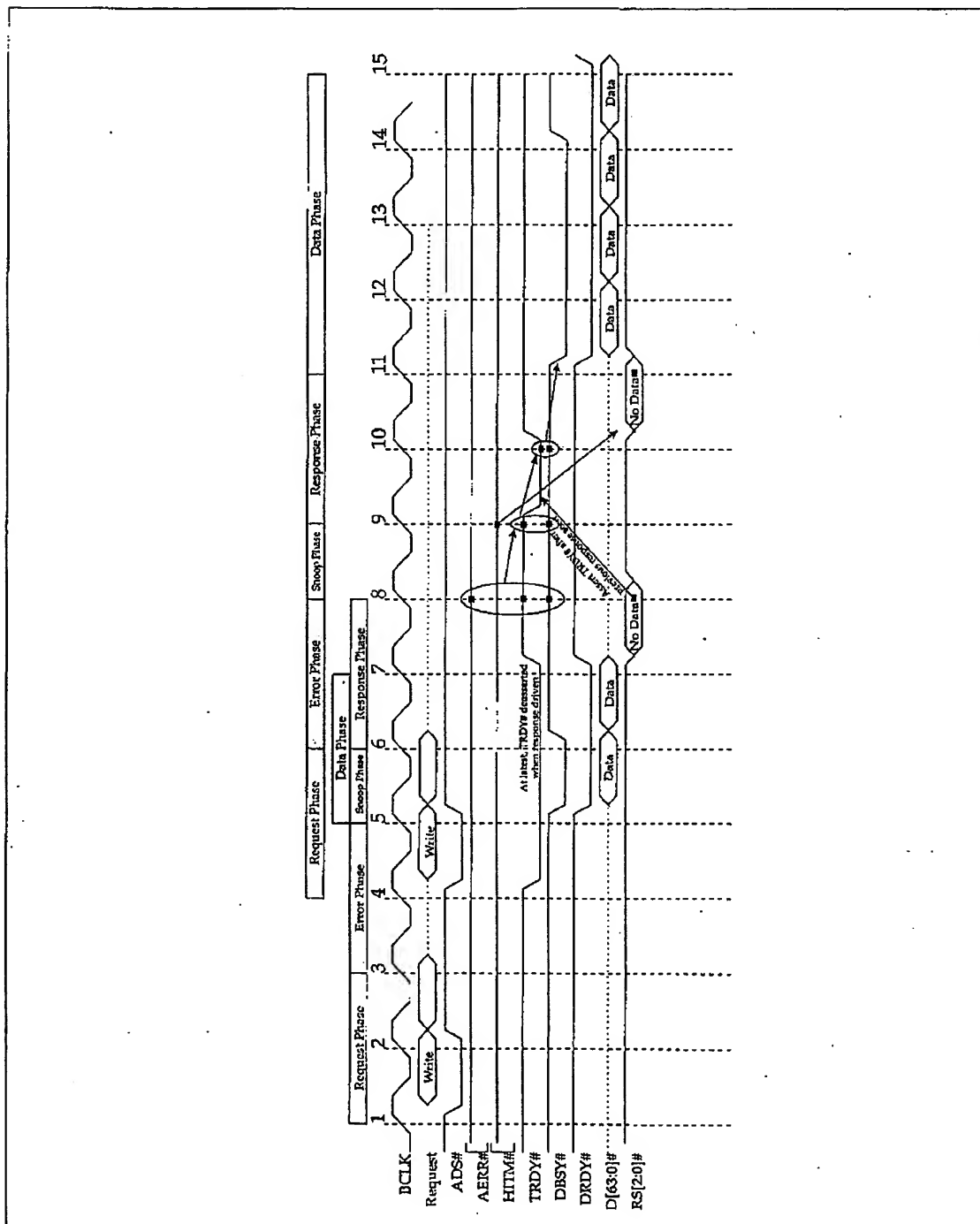
Figure 13-3 on page 288 illustrates two transactions:

- a two-quadword (16 byte) write that doesn't hit a modified line and isn't deferred.
- a four-quadword (32-byte) write that doesn't hit a modified line and isn't deferred.

In both cases, the request agent writes data to the response agent in the data phase of the transaction. The following discussion focuses on the second of these two transactions.

Pentium Pro Processor System Architecture

Figure 13-3: Write that Doesn't Hit a Modified Line and Isn't Deferred



Chapter 13: The Response and Data Phases

Basics

1. When it is ready to receive the write data from the request agent, the response agent asserts TRDY#.
2. When the request agent samples TRDY# asserted (the response agent is ready to receive the write data) and, if the previous transaction required a data transfer, DBSY# has been deasserted (the data bus is free for it to use), it asserts DBSY# in preparation for driving the write data to the response agent.
3. The response agent supplies the No Data response in the response phase.
4. None of the snoop agents has a copy of the line in the modified state and therefore a snooper will not be writing data to memory.

Previous Transaction May Involve a Write

On a write transaction, the response agent asserts TRDY# when it is ready to receive the write data from the request agent. It must make sure that it doesn't start driving TRDY# too early, however. As shown in Figure 13-3 on page 288, the previous transaction may involve a write as well (either because it's a write transaction or because it hits on a modified line and the snoop agent writes the modified line to memory). In this case, the response agent of the previous transaction asserts TRDY# when it's ready to accept the write data. After being asserted, TRDY# is deasserted, at the latest, when a transaction's response is driven (see clock 7).

Earliest TRDY# Assertion is 1 Clock After Previous Response Issued

In the current transaction, it's therefore definitely safe to assert TRDY# (if the response agent is ready to accept write data) one clock after the previous response agent provides its response (see clock 9). By then, if TRDY# had been asserted by the previous transaction's response agent, it has been deasserted and has quit ringing.

When Does Request Agent First Sample TRDY#?

On a write transaction, the request agent first samples TRDY# at the end of the error phase (see clock 8) at the same time that it samples AERR#. If the transaction isn't cancelled by the assertion of AERR#, it will proceed. The earliest at which TRDY# should therefore be asserted (assuming that the response for the previous transaction has been seen—see previous two sections) is three clocks after ADS# is asserted (in other words, during the second clock of the transaction's error phase, clock 7).

Pentium Pro Processor System Architecture

When Does Request Agent Start Using Data Bus?

When the request agent samples AERR# deasserted (see clock 8), the transaction proceeds. The request agent starts sampling TRDY# and DBSY# at the end of the error phase at the same time that it samples AERR#. There are three cases:

1. Request agent samples TRDY# asserted and DBSY# deasserted—This is not the case in the figure. AERR# and DBSY# are deasserted when first sampled on clock 8, but TRDY# isn't asserted until clock 10. The request agent will therefore continue to sample TRDY# and DBSY# on each clock until TRDY# is asserted and DBSY# is deasserted. This indicates that the response agent is ready to receive the data (TRDY# asserted) and the data bus is not in use (DBSY# deasserted). When these conditions are met, the request agent can take ownership of the data bus (assert DBSY#) one clock later to start driving the write data to the response agent.
2. Request agent samples TRDY# asserted and DBSY# asserted—Although the response agent is ready to accept the data (TRDY# asserted), the agents involved in the previous transaction are still using the data bus (DBSY# still asserted). In this case, the request agent must not take ownership of the data bus until it samples TRDY# asserted and DBSY# deasserted. It may then take ownership of the data bus (assert DBSY#) one clock later.
3. Request agent samples DBSY# deasserted and TRDY# deasserted (see clocks 8 and 9)—Although the data bus is free, the response agent is not yet ready to accept the data. The request agent must not take ownership of the data bus until it samples TRDY# asserted and DBSY# deasserted (see clock 10).

When Can TRDY# Be Deasserted?

Once asserted, the earliest at which TRDY# can be deasserted is when the response agent is sure that the request agent has seen TRDY# asserted and DBSY# asserted (so it can take ownership of the data bus).

If the target samples DBSY# deasserted on the same clock that it starts asserting TRDY# (see clock 9), it knows that the request agent will see TRDY# asserted and DBSY# deasserted on the next clock (see clock 10). The response agent can therefore turn off TRDY# one clock after it asserts it (see clock 10). This one clock assertion of TRDY# may be done as long as it's not sooner than three clocks from the previous deassertion of TRDY#.

If the response agent doesn't sample DBSY# deasserted when it starts asserting TRDY# (this is not the case in clock 9 of the figure), it must keep TRDY# asserted until it samples DBSY# deasserted. When the response agent sees this,

Chapter 13: The Response and Data Phases

it can deassert TRDY#. It knows that the request agent has seen it as well and that it will take ownership of the data bus one clock later (see clock 11). At the latest, the target must deassert TRDY# when it delivers the No Data response (see clock 10).

When Does Request Agent Take Ownership of Data Bus?

The request agent takes ownership of the data bus one clock after seeing TRDY# asserted (response agent is ready to take the data) and DBSY# deasserted (previous data bus owner has relinquished ownership). This occurs in clocks 10 and 11 of the figure.

Deliver the Data

As the request agent drives each of the quadwords onto the bus, it asserts DRDY# to indicate its presence. In the figure, this occurs on clocks 11 through 14. The response agent latches the data from the bus on clocks 12 through 15 along with the state of the DRDY# signal. DRDY# asserted indicates that the data latched is valid. In clock 14, the request agent deasserts DBSY# to relinquish data bus ownership. In clock 15, it deasserts DRDY# as well (the final data item has been transferred).

On AERR# or Hard Failure Response

If the request agent receives an AERR# in the error phase or a hard failure response in the response phase, it may have already accepted some write data from the request agent. If it has, the response agent discards the data. In other words, it is important that data written to a response agent should initially be accepted into a buffer and only written to the target memory or IO location(s) if there's no AERR# and no hard failure response.

Snoop Agents Change State of Line from E->I or S->I

If any snoop agent had a hit on a line in the E state, it changes it to the I state (because it cannot snarf data being written to memory by another request agent). If any of the snoop agents had a copy of the line in the S state, they change it to the I state.

Read that Hits a Modified Line

Figure 13-4 on page 293 illustrates a read that hits on a modified line. Refer to this figure during the following discussion. The read could be for any amount of data up to a full line.

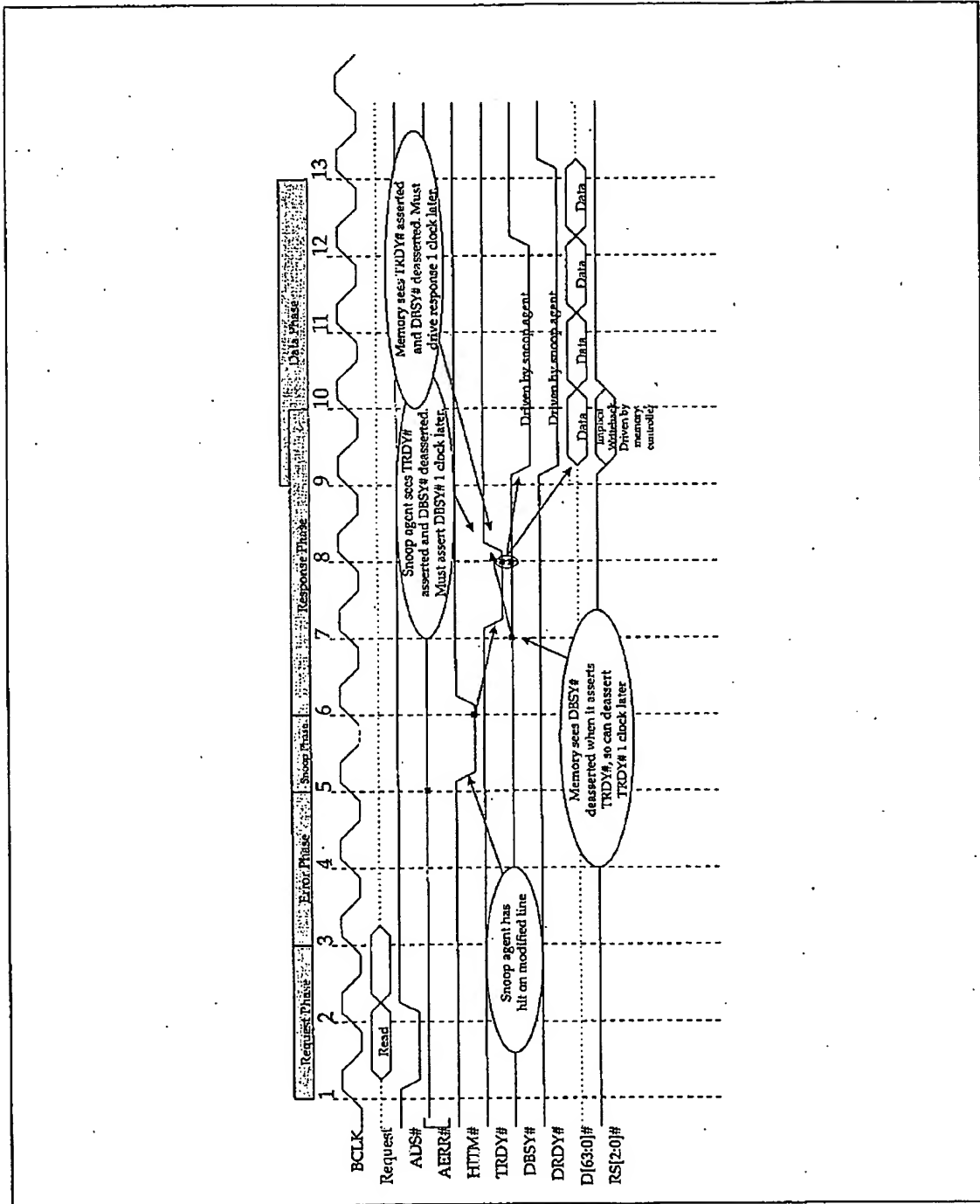
Pentium Pro Processor System Architecture

Basics

1. The request agent starts a read transaction to read data from memory.
2. The snoop results in the assertion of HITM#, indicating that a snoop agent has a copy of the requested line in the modified state.
3. The HITM# indication informs the response agent (i.e., the memory controller) that it does not have a fresh copy of the requested line. Rather the snoop agent has it and will supply the line to memory and to the request agent.
4. The transaction is still a read from the request agent's standpoint, but from the memory controller standpoint has changed from a read by the request agent into a write to memory by the snoop agent.
5. The memory controller must provide the Implicit Writeback response in the response phase.
6. When the memory controller is ready to accept the modified line (in other words, it has a 32-byte buffer available), it asserts TRDY# to the snoop agent.
7. The snoop agent is waiting to see TRDY# asserted (memory is ready to accept the write data) and DBSY# deasserted (the data bus isn't busy). It keeps sampling both signals until this state is seen. It then asserts DBSY# to take ownership of the data bus in preparation for writing the data to memory.
8. The modified line is then provided to memory and to the request agent. The four quadwords are provided in toggle mode order, critical quadword first. If the request agent has only requested a subset of the quadword, it just takes what it asked for and ignores the remaining quadwords. The memory must accept the entire line.
9. The snoop agent has freshened up memory, so it changes the state of the line in its cache from modified to shared (because it has just watched someone else read from the line).

Chapter 13: The Response and Data Phases

Figure 13-4: Read that Hits a Modified Line



Pentium Pro Processor System Architecture

Transaction Starts as a Read from Memory

The request agent issues a memory read request in the request phase (starting on clock 1). Assuming that the transaction isn't cancelled by the assertion of AERR# in the error phase (see clock 5), the snoop phase is entered and the snoop agent that has a copy of the requested line in the modified state asserts HITM# (see clock 5).

From Memory Standpoint, Changes from Read to Write

The assertion of HITM# (see clock 6) informs the memory controller that its copy of the requested line is stale and that the snoop agent will supply the modified line to memory and to the request agent in the data phase of the transaction. The transaction remains a read from the request agent's standpoint.

Memory Asserts TRDY# to Accept Data

Assuming that the previous transaction's response has already been seen (so TRDY#, if it had been asserted, has been deasserted) and that memory is ready to accept the data, the memory controller asserts TRDY# to the snoop agent (see clock 7). If DBSY# is in the deasserted state (indicating that the previous data bus owner has relinquished ownership) on the clock that TRDY# is driven asserted, the memory controller can deassert TRDY# one clock after it asserts it (because it is assured that the snoop agent sees TRDY# asserted and DBSY# deasserted one clock after TRDY# is asserted). Otherwise, it must keep TRDY# asserted until it samples DBSY# deasserted.

After indicating the HITM#, the snoop agent starts sampling TRDY# and DBSY# two clocks after the snoop phase ends (see clock 8) to determine when it can take ownership of the data bus. The response agent must be ready to accept the data (TRDY# asserted) and the previous data bus owner must have relinquished ownership (DBSY# deasserted).

Memory Must Drive Response At Right Time

The memory controller must drive the Implicit Writeback response (on clock 9) one clock after sampling TRDY# asserted and DBSY# deasserted (see clock 8). This is necessary because:

- On a read, DBSY# must be asserted at the same time that the response is driven.
- When the snoop agent samples TRDY# asserted and DBSY# deasserted, it will assert DBSY# one clock later in preparation for driving the write data.

Chapter 13: The Response and Data Phases

Snoop Agent Asserts DBSY# and Memory Drives Response

As stated earlier, when the snoop agent sees TRDY# asserted and DBSY# deasserted (see clock 8), it takes ownership of the data bus one clock later by asserting DBSY# (see clock 9). At the same time, the memory controller drives the Implicit Writeback response to the request agent.

Snoop Agent Supplies Line to Memory and to Request Agent

After asserting DBSY#, the snoop agent writes the line to memory and to the request agent. As it provides each of the four quadwords, it asserts DRDY# to indicate the presence of the quadword. On a read, the request agent starts sampling DRDY# on the clock that the response is received (clock 10) and latches a quadword each time that DRDY# is sampled asserted. Note that if the read is only for one or two quadwords and it hits a modified line (implicit writeback response), the request agent will only take the quadwords that it requested and will ignore the remaining quadwords.

Snoop Agent Changes State of Line from M->S

When memory has received the fresh line, the line is no longer modified in the snoop agent's cache, so it changes it from the M to the S state (because it knows the request agent read it and, if it has a cache, has placed the line in the S state).

Write that Hits a Modified Line

Figure 13-5 on page 297 illustrates a two-quadword memory write that hits a modified line in the cache of one of the snoop agents' caches. Refer to this figure during the following discussion.

1. Transaction starts as a write from the request agent to the response agent (i.e., the memory controller).
2. The response agent asserts TRDY# in clock 4 and samples DBSY# deasserted when it starts asserting TRDY#. This means that the request agent will see TRDY# asserted and DBSY# deasserted on the next clock (clock 5), giving it permission to take ownership of the data bus in clock 6. The response agent can therefore deassert TRDY# one clock after it asserts it.
3. AERR# isn't asserted in the error phase (see clock 5), so the transaction proceeds.
4. At the end of the error phase, AERR# is sampled deasserted and the request agent samples TRDY# for the first time to see if the memory controller is ready to accept the write data. In addition, DBSY# is sampled to see if the data bus is still in use by the previous transaction's agents. The request

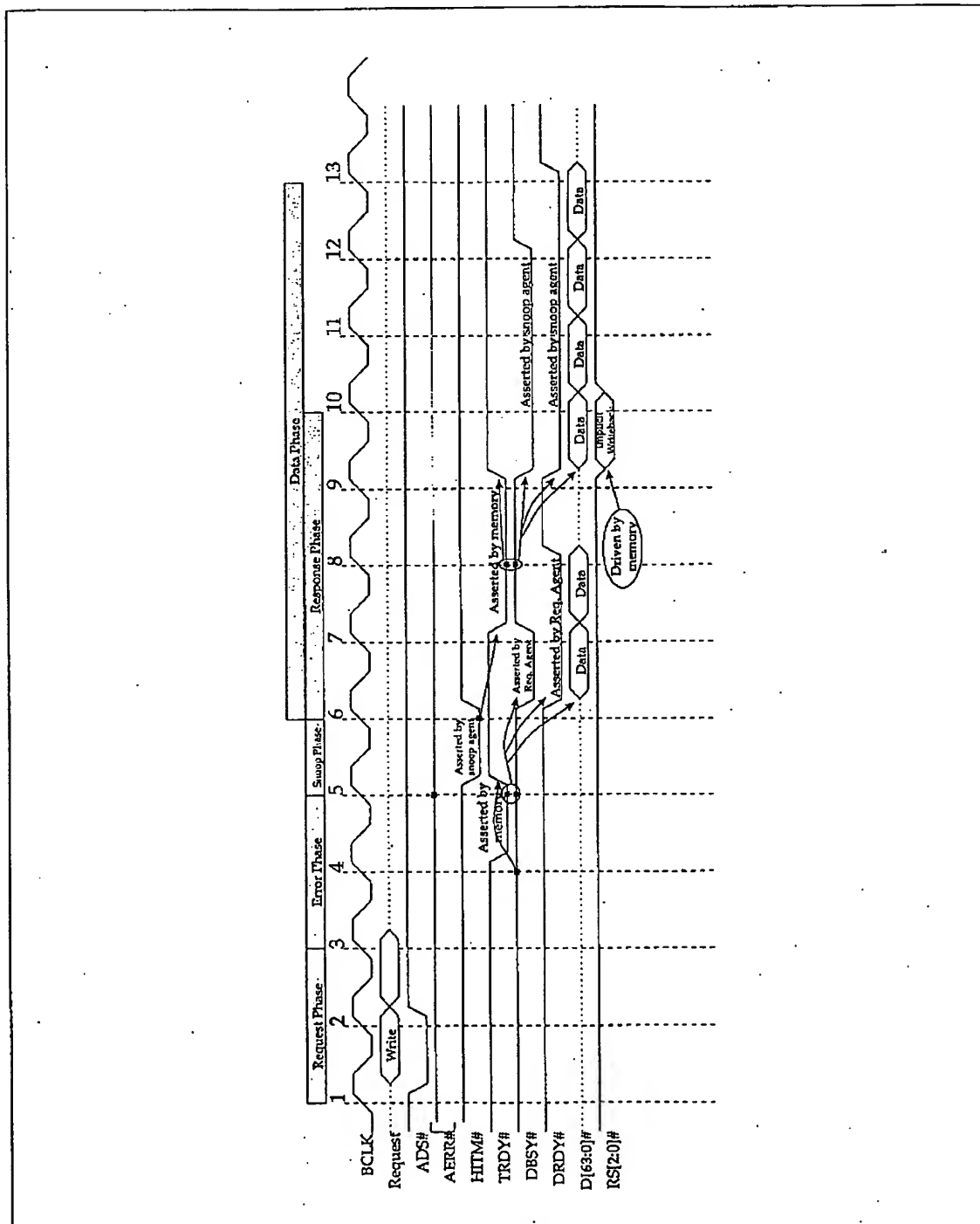
Pentium Pro Processor System Architecture

agent doesn't acquire data bus ownership until the clock where it samples TRDY# asserted and DBSY# deasserted (see clock 5).

5. In the snoop phase, a snoop agent that has a copy of the target line in the modified state asserts HITM# (see clock 6). This informs the memory controller that it will receive write data from two sources—first the write data from the request agent and then the modified line from the snoop agent.
6. When TRDY# has been sampled asserted and DBSY# is sampled deasserted (see clock 5), the request agent asserts DBSY# (see clock 6), taking ownership of the data bus in preparation for writing its data to the memory controller.
7. The request agent drives out the first of the two quadwords in clock 6 and asserts DRDY# to indicate its presence on the data bus. The response agent samples the data and DRDY# on clock 7. DRDY# is sampled asserted, qualifying the data just latched as valid. In clock 7, the request agent drives the second quadword onto the data bus and keeps DRDY# asserted to indicate its presence. It also deasserts DBSY#, relinquishing ownership of the data bus. The response agent samples the data and DRDY# on clock 8. DRDY# is sampled asserted, qualifying the data just latched as valid. All of the data has been written to the response agent.
8. When the memory controller samples TRDY# asserted and DBSY# deasserted (see clock 5), the memory controller deasserts TRDY# for at least one clock and then reasserts it (in clock 7). The second assertion of TRDY# informs the snoop agent that the memory controller is ready to accept the modified line (in other words, it has a 32-byte buffer available to receive the line).
9. When the snoop agent samples TRDY# asserted for the second time (in clock 8), it also samples DBSY# to see if the request agent has surrendered ownership of the data bus yet. When it samples DBSY# deasserted (see clock 8), it asserts DBSY# (see clock 9) to take ownership of the data bus.
10. When the memory controller has asserted TRDY# for the second time and then samples DBSY# (see clock 8) deasserted (indicating that the request agent is through with its write), it deasserts TRDY# (see clock 9) and drives the Implicit Writeback response to the request agent.
11. Starting in clock 9, the snoop agent uses the data bus to write the line to memory. As it drives each of the four quadwords onto the data bus, it asserts DRDY# to indicate its presence. As it drives the final quadword onto the bus (in clock 12), it deasserts DBSY# to relinquish data bus ownership. When it has transferred the final quadword (on clock 13), it deasserts DRDY#.
12. In addition, the snoop agent invalidates its copy of the line (because it cannot snarf the data written to memory by the request agent to keep its copy up to date).

Chapter 13: The Response and Data Phases

Figure 13-5: Write that Hits a Modified Line



Pentium Pro Processor System Architecture

Data Phase Wait States

The agent sourcing the data can delay delivery of each quadword by keeping $\overline{\text{RDY\#}}$ deasserted until it presents each quadword. As it drives a quadword onto the bus, it asserts $\overline{\text{RDY\#}}$ to indicate its presence on the bus.

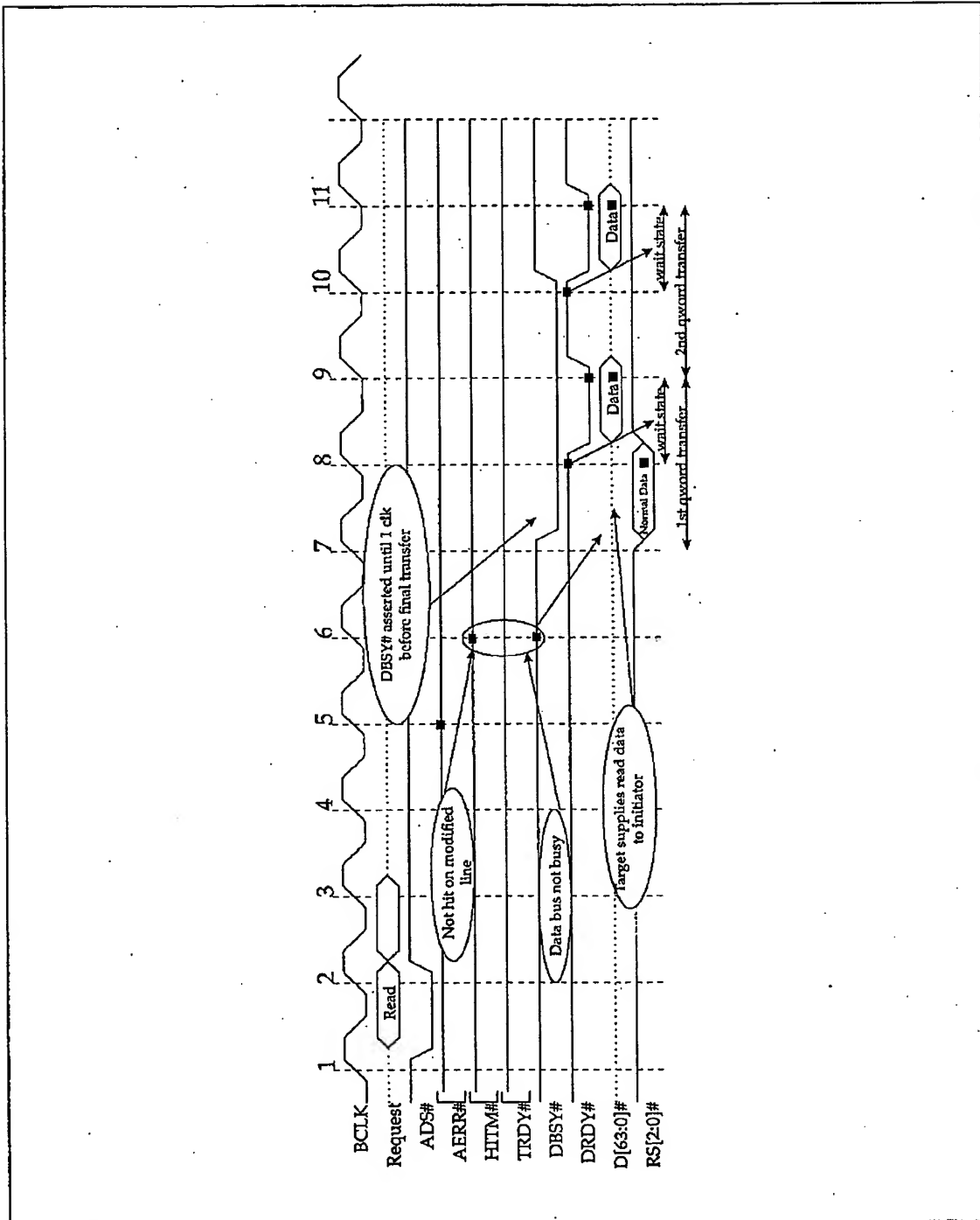
Figure 13-6 on page 299 illustrates a two-quadword read where the response agent takes two clocks (one wait state) to deliver each quadword.

Figure 13-7 on page 300 illustrates a four-quadword write wherein the request agent delivers the first quadword in 0-wait states, and inserts one wait state into the delivery of each of the subsequent three quadwords.

The specification doesn't place a limit on the number of wait states that may be inserted into the transfer of a quadword. However, the system designer may choose to monitor the behavior of agents to ensure that none of them inserts excessive wait states. This would adversely affect all subsequently-issued transactions that require the use of the data bus.

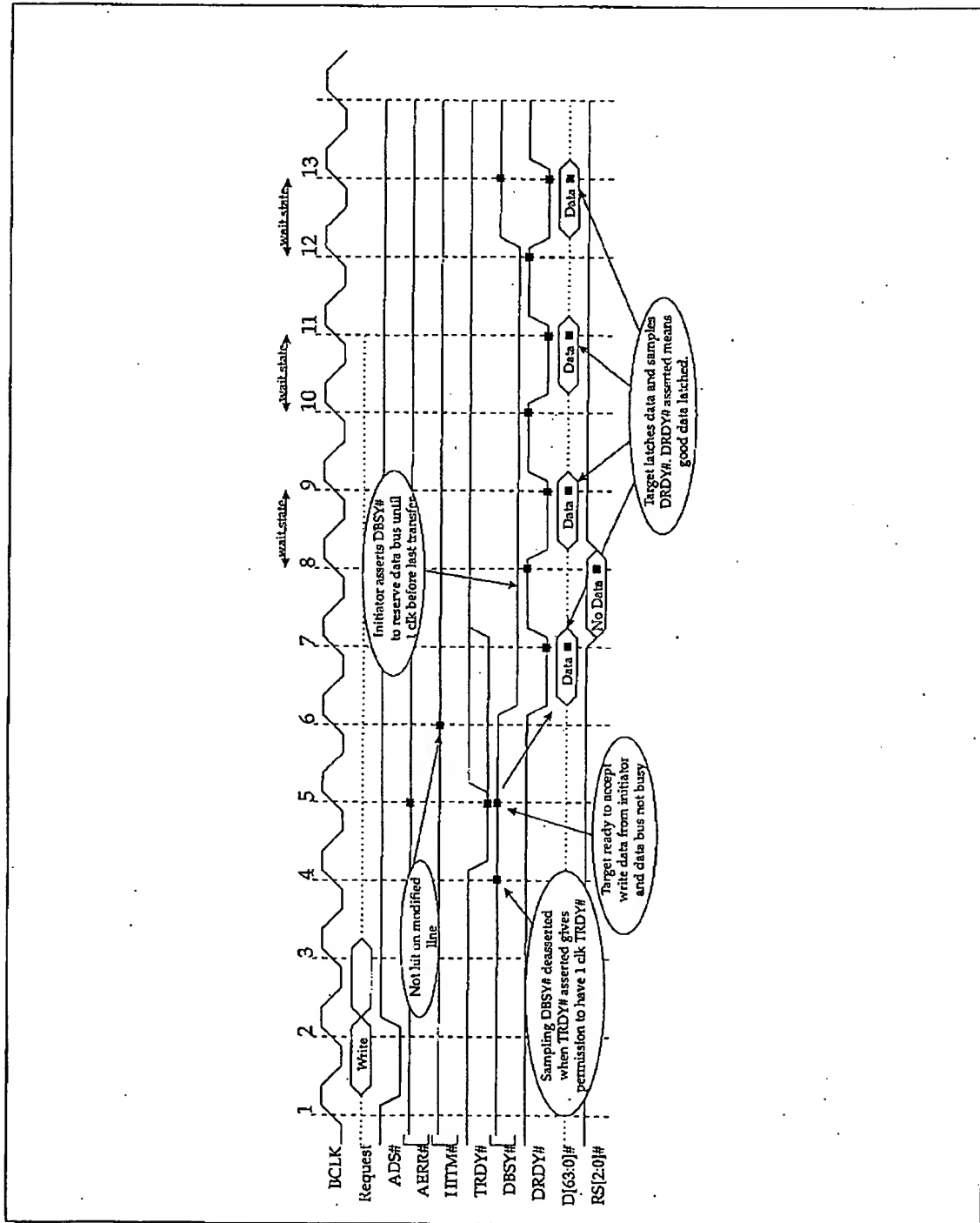
Chapter 13: The Response and Data Phases

Figure 13-6: Example of Two-Quadword Read with Wait States



Pentium Pro Processor System Architecture

Figure 13-7: Example of Four Quadword Write with Wait States



Chapter 13: The Response and Data Phases

Special Case—Single Quadword, 0-Wait State Transfer

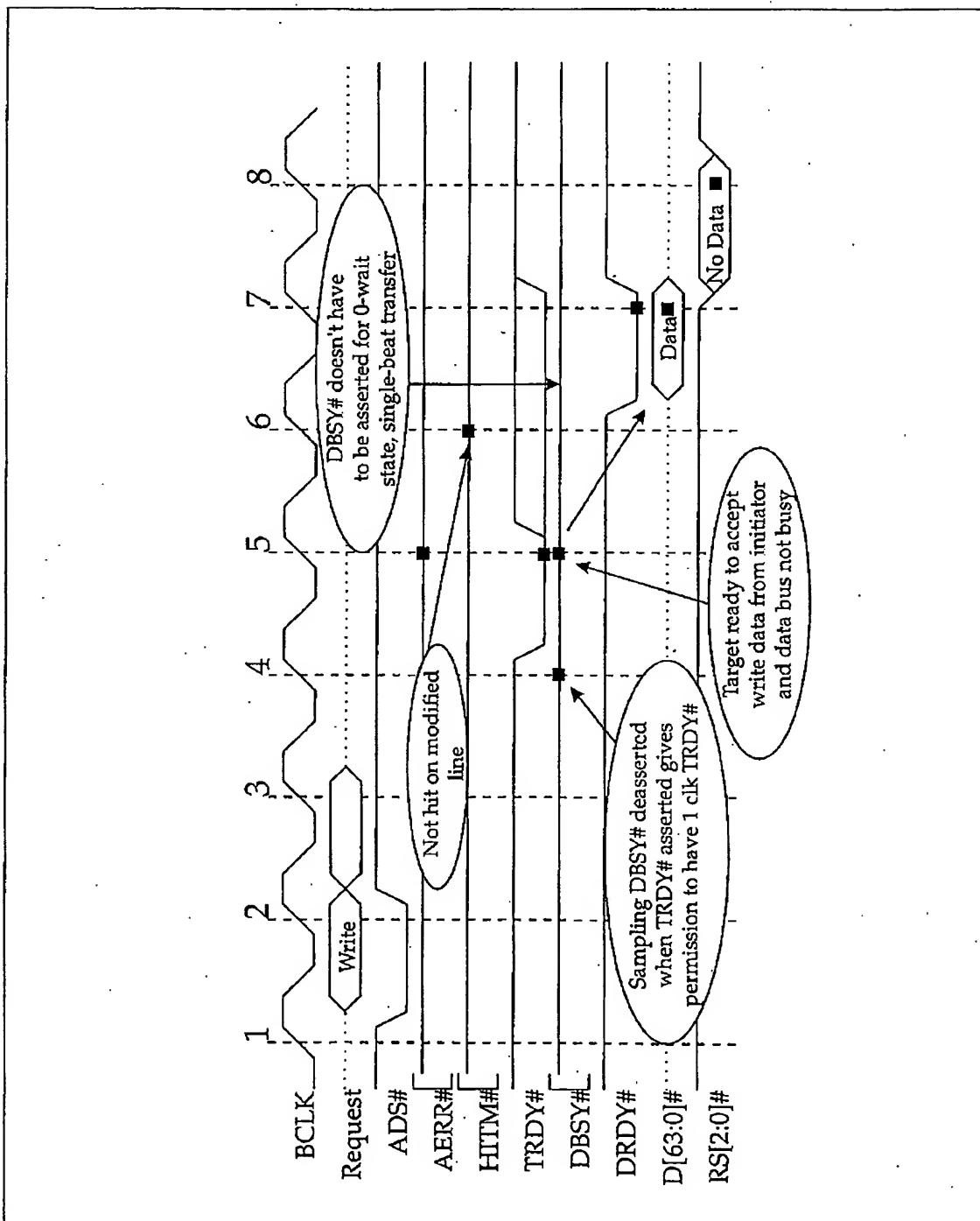
Figure 13-8 on page 302 illustrates a single-quadword, 0-wait state write. When a transaction only transfers one quadword (or a subset thereof), and the agent sourcing the data can immediately supply the data when it gains data bus ownership, the agent does not have to assert DBSY#. It just asserts DRDY# immediately upon acquiring data bus ownership and drives out the quadword. In Figure 13-8 on page 302:

1. The request agent samples TRDY# asserted and DBSY# deasserted on clock 5 and takes ownership of the data bus on clock 6.
2. The request agent asserts DRDY# immediately upon acquiring data bus ownership (on clock 6) and drives out the quadword.
3. The response agent samples the data and the state of DRDY# on clock 7. DRDY# is sampled asserted, qualifying the data as valid.

There is no danger of an ownership collision on the data bus. Whether the next transaction is a read or a write, the agent that will source data cannot take ownership of the data bus until the response for this transaction has been seen.

Pentium Pro Processor System Architecture

Figure 13-8: Example of Single-Quadword, 0-Wait State Write



Chapter 13: The Response and Data Phases

Response Phase Parity

The RS[2:0]# signals are protected by the RSP# parity bit. During the response phase, it must be left high or driven low to force an even number of electrical lows in the overall 4-bit pattern. Note that if the response agent stalls the response phase (because it doesn't have the response ready to be delivered yet), proper parity must be provided for each of idle periods until the actual response is driven.

Hardware

Section 4:

Other Bus Topics

The Previous Section

The chapters that comprise Part 2, Section 3 focused on the phases a transaction passes from inception to completion.

This Section

The chapters that comprise Part 2, Section 4 cover additional transaction and bus topics not covered in the previous sections. This is the final section of Part 2 and concludes the discussion of the processor's hardware characteristics. It consists of the following chapters:

- "Transaction Deferral" on page 307.
- "IO Transactions" on page 329.
- "Central Agent Transactions" on page 333.
- "Other Signals" on page 345.

The Next Part

Part 3 focuses on the software characteristics of the Pentium Pro processor.

14 *Transaction Deferral*

The Previous Chapter

The previous chapter concluded “Hardware Section 3: The Transaction Phases.” It provided a detailed description of the response and data phases of a transaction.

This Chapter

This chapter is the first chapter of “Hardware Section 4: Other Bus Topics” and provides a detailed description of transaction deferral and the deferred reply transaction.

The Next Chapter

The next chapter provides detailed information regarding IO read and write transactions.

Introduction to Transaction Deferral

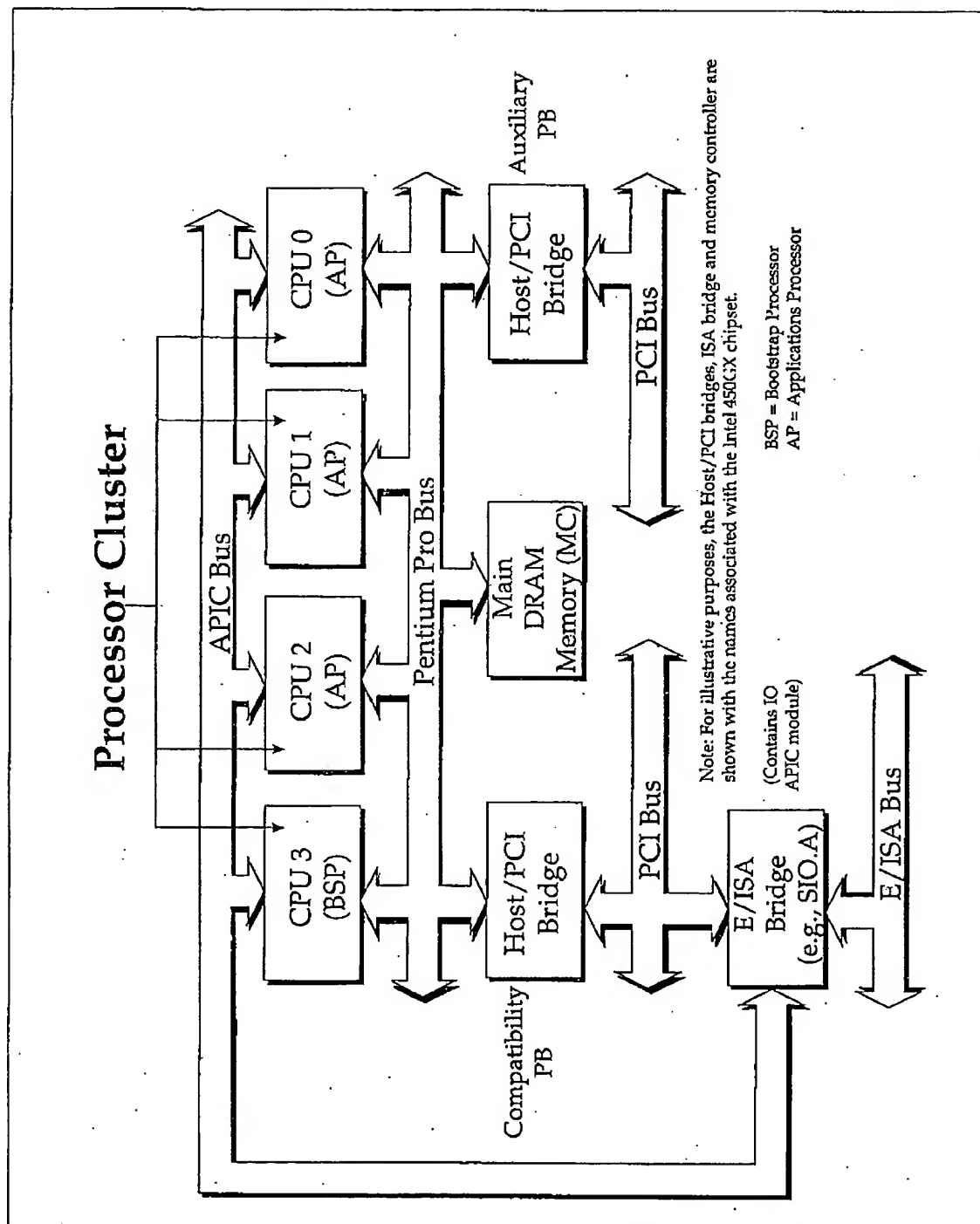
For an introduction to the topic of transaction deferral, refer to the discussion that starts with “Transaction Retry and Deferral” on page 268. After reading that discussion, then proceed with the remainder of this chapter.

Example System Model

The sections in this chapter describe deferred transactions and deferred reply transactions using the example system model pictured in Figure 14-1 on page 308.

Pentium Pro Processor System Architecture

Figure 14-1: Example System with one Pentium Pro Bus and Two Host/PCI Bridges



Chapter 14: Transaction Deferral

Typical PC Server Model

This discussion focuses on the use of transaction deferral and deferred reply transactions to increase the overall performance of the system pictured in Figure 14-1 on page 308. Please note that there are some references to PCI bus operation. For a detailed description of the PCI bus protocol, refer to the MindShare book entitled *PCI System Architecture* (published by Addison-Wesley).

The Problem

When any of the processors attempts to perform a read or write with a target residing on either of the PCI buses or on the EISA or ISA bus, it can result in very long latency during the data phase of the transaction.

When the transaction is initiated, all of the agents on the processor bus latch the transaction and each of the response agents examines the address and transaction type to determine which of them is the targeted response agent. Assuming that the processor is targeting a device that resides beyond one of the host/PCI bridges, that bridge must act as the response agent for the transaction. Essentially, it is the surrogate for the addressed target which resides somewhere on the other side of the bridge. If the transaction is a read, the bridge takes ownership of the processor data bus (by asserting DBSY#), but keeps DRDY# deasserted until it has the requested read data. It then presents the data to the processor and asserts DRDY# to indicate its presence.

Before the bridge can provide the requested read data from the target or can deliver the write data to the target, it must first arbitrate for ownership of the PCI bus. When it asserts its REQ# to the PCI bus arbiter, the following conditions may be true:

- the PCI bus is currently in use by a PCI master performing a burst read or write transaction with another device on the PCI bus.
- one or more other bus masters that reside on the PCI bus may also be asserting their REQ# to the PCI bus arbiter.
- the host/PCI bridge may have been the last owner of the PCI bus and the arbiter may use a rotational priority scheme.

If all of the above conditions are true, the host/PCI bridge will not attain ownership of the PCI bus until each of the other PCI masters has each received ownership of the PCI bus. Upon attaining ownership, each master may then retain ownership of the bus until it has exhausted its assigned timeslice (i.e., master

Pentium Pro Processor System Architecture

latency timer, or MLT). When the host/PCI bridge finally attains ownership (that could be quite a while) and initiates the PCI read or write transaction, it must then wait up to five PCI clocks for the PCI target to claim the transaction (i.e., assert DEVSEL#). Once the target has claimed the transaction, it will issue a retry to the bridge if the transaction will take more than 16 clocks from its start (the assertion of FRAME#) to transfer (read or write) the first dword. This 16 clock rule was added in the 2.1 PCI specification. The target has memorized the transaction and is processing it off-line. The host bridge must then periodically retry the transaction until the target finally hands over the read data or indicates that it has accepted the write data. The absolute worst-case scenario occurs if the targeted device resides on the ISA bus (an extremely slow, 8MHz bus populated by horrendously slow devices).

If the host/bridge had kept DRDY# deasserted on the processor bus during this entire period of time, the data bus would remain busy (DBSY# asserted). This means that all transaction requests (that require a data transfer) subsequently issued by any bus agents (including the same one) will stall.

The result—gridlock!

Possible Solutions

The designers of the host/PCI bridge can take one of three possible approaches:

1. **Hang the bus** for extensive periods of time (as described in the previous section). This is certainly the least-desirable approach.
2. The host/PCI bridge can **memorize the transaction and issue a retry response** to the processor. This obligates the processor to re-request the processor bus and retry the transaction on a periodic basis until it gets a good response and the read or write completes. The bridge then arbitrates for the PCI bus and proceeds as described in the previous section. When the bridge has finally completed the requested transaction of the PCI side; it waits for the processor's next retry. When it latches a transaction issued on the processor side, it compares the agent and transaction IDs to see if it matches the transaction that was issued a retry response earlier. When it has a match, it permits the transaction to complete—if it's a read, it supplies the read data that it obtained from the PCI side and a Normal Data response; if it's a write, it accepts the data and issues a No Data response. Although better than option one, the retried processor's repeated intrusions into the symmetric arbitration and its usage of the request, AERR#, snoop, and response signal groups will significantly diminish the performance of the other processors. *This is the approach used by the Intel 450GX and 450KX host/PCI bridges.*

Chapter 14: Transaction Deferral

3. The optimal approach is for the host/PCI bridge to memorize the transaction and issued the **deferred response** to the processor. The processor will not retry the transaction. Rather, it will wait for the response agent to initiate a **deferred reply** transaction to provide the completion notice and, if a read, the read data. The processor therefore doesn't waste valuable bus time with fruitless re-issues of the transaction and the bus remains available for the processors to use (including this processor if MTRRs permit out-of-order execution in the area addressed by the transaction). *This is the approach used by the Intel 440FX bridge.*

An Example Read

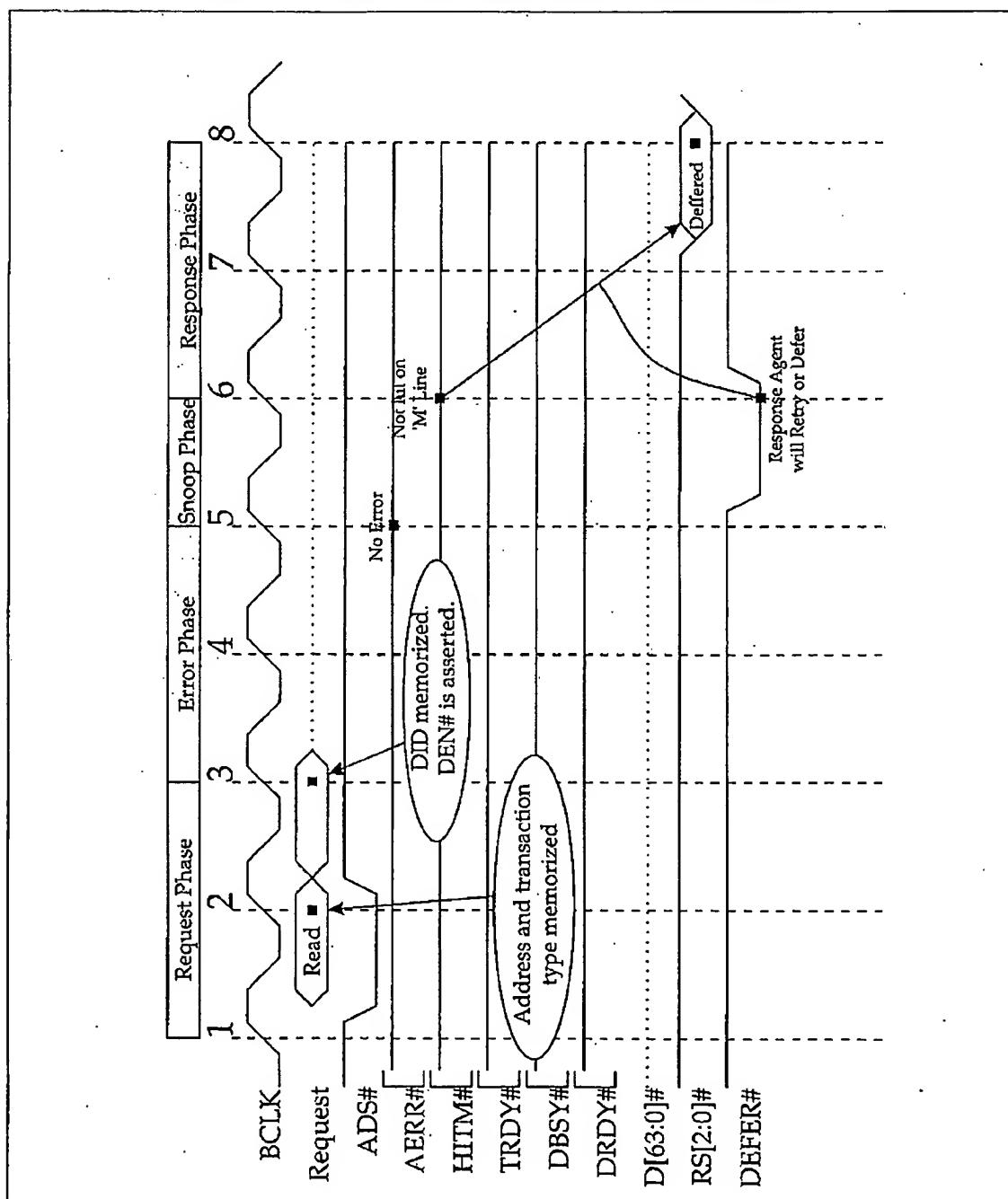
Read Transaction Memorized and Deferred Response Issued

Refer to Figure 14-1 on page 308 and Figure 14-2 on page 312 during this example. Assume that a processor initiates a read transaction that targets a device residing beyond one of the host/PCI bridges:

1. The bridge memorizes the transaction, including the Deferred ID, or DID, delivered in request packet B.
2. This example assumes that the request initiator permits the response agent to defer the transaction (DEN# is asserted in request packet B).
3. The bridge asserts DEFER# to the request agent in the snoop phase, indicating that it intends to issue a retry or a deferred response in the response phase.
4. In the response phase of the transaction, the bridge acts as the response agent and issues the deferred response. This causes the request agent to mark the transaction request as deferred and it moves it from its IOQ to its deferred transaction queue. All other agents remove it from their IOQs and cease tracking it. Effectively, the read transaction is now in a state of suspension until, at a later time, the response agent (i.e., the bridge) addresses the request agent using a deferred reply transaction.
5. The data phase of the transaction has been deferred and will be performed during the deferred reply transaction that will be initiated by the bridge at a later time (when it has obtained the read data).

Pentium Pro Processor System Architecture

Figure 14-2: Read Transaction Receives a Deferred Response



Chapter 14: Transaction Deferral

Bridge Performs PCI Read Transaction

As discussed earlier, the bridge performs the equivalent PCI read transaction to obtain the requested data from the target device. The PCI transaction completes in one of the following ways:

1. the transaction completes successfully and the requested data is read from the target and stored in a temporary buffer in the bridge. In this case, the bridge must indicate the normal data response in the response phase of the deferred reply transaction.
2. the transaction results in a target abort from the target, indicating that it is broken. In this case, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
3. the transaction ends in a master abort because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed). In this case, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
4. Read parity error detected. In this case, the bridge may re-attempt the PCI transaction to see if the data can be read successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction). If it still fails or if no re-attempt is made, the bridge must indicate one of the following responses in the response phase of the deferred reply transaction:
 - the hard failure response, or
 - the normal data response, but pass the bad data and parity to the request agent as is.

Deferred Reply Transaction Issued

Refer to Figure 14-3 on page 315 during the following discussion. When the PCI transaction has completed, the bridge uses BPRI# to arbitrate for ownership of the processor bus. Be aware that, in Figure 14-1 on page 308, the two host/PCI bridges both use the BPRI# signal to request ownership of the processor bus, but only one is permitted to drive it at a time. Before asserting BPRI#, therefore, if both of the bridges require ownership of the processor bus, they must use sideband signals to arbitrate between themselves for ownership of the BPRI# signal. The winner then asserts BPRI# to gain ownership of the request signal group.

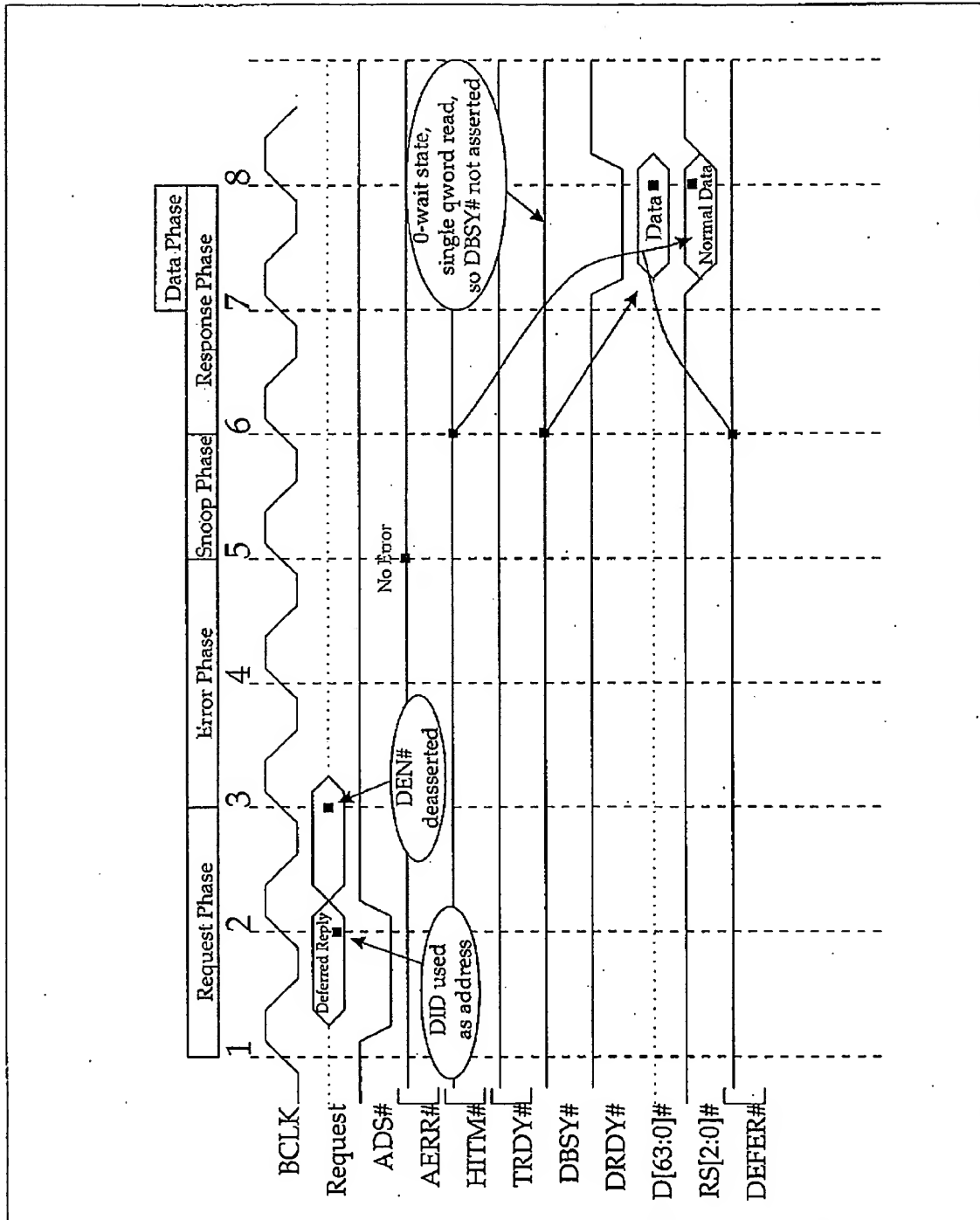
Having acquired ownership of the request signal group, the bridge then issues a deferred reply transaction. During the request and error phases, the bridge acts as the request agent and the processor addressed by the deferred ID (see

Pentium Pro Processor System Architecture

Table 14-1 on page 316) acts as the target of the transaction. The information indicated in Table 14-1 on page 316 is driven out during the transmission of packets A and B. All agents on the bus latch the two packets during the request phase, check parity, and issue AERR# in the error phase if there's a problem. If there isn't a problem, the transaction proceeds to the snoop phase.

Chapter 14: Transaction Deferral

Figure 14-3: Deferred Reply Transaction for Read



Pentium Pro Processor System Architecture

Table 14-1: Request Packets A and B Content in Deferred Reply Transaction

Signals	Description
Request Packet A Content	
REQ[4:0]#	00000b = Deferred reply transaction type.
A[35:24]#	Don't care, but must be in a stable state because the address parity, AP[1:0]#, must be correct.
A[23]#	DID[7]#. Request agent type.
A[22:20]#	DID[6:4]#. Agent ID.
A[19:16]#	DID[3:0]#. Transaction ID.
A[15:3]#	Don't care, but must be in a stable state because the address parity, AP[1:0]#, must be correct.
AP[1:0]#	Address parity bits must be correct (see Table 11-3 on page 246).
RP#	REQ[4:0]# parity bit, RP#, must be correct (see Table 11-3 on page 246).
Request Packet B Content	
SPLCK#	Split Lock is output on the A[6]# pin and must be deasserted.
DEN#	Defer Enable is output on the A[4]# pin and must be deasserted (the deferred reply transaction may not be deferred).
A[35:3]#	Don't care, but must be in a stable state because the address parity, AP[1:0]#, must be correct.
REQ[4:0]#	Don't care, but must be in a stable state because the REQ[4:0]# parity bit, RP#, must be correct.
AP[1:0]#	Address parity bits must be correct (see Table 11-7 on page 250).
RP#	REQ[4:0]# parity bit, RP#, must be correct (see Table 11-7 on page 250).

Chapter 14: Transaction Deferral

Original Request Agent Selected

All of the agents match the DID field latched in packet A (from A[23:16]#) to identify a previously-deferred transaction that has been in a state of suspension in their deferred transaction queues. The request agent that originated the read has a match on its agent type, agent ID, and transaction ID.

Bridge Provides Snoop Result

Until the snoop phase, the bridge is the request agent of the deferred reply transaction, but an interesting thing happens in the snoop phase. The deferred reply transaction is never snooped by the processor bus snoop agents. Rather, the snoop result is supplied by the bridge (i.e., the agent that deferred the transaction earlier) with the snoop result obtained from caches on the PCI bus when it did the read. Since typical system designs do not permit PCI agents to cache memory information, the snoop result delivered on HIT# and HITM# indicates a cache miss (both are deasserted). In addition, the DEFER# signal is not asserted (DEN# is always deasserted in request packet B during a deferred reply transaction).

Response Phase—Role Reversal

Once the snoop result is delivered to the processor by the bridge, the response phase of the transaction is entered. The role reversal is now complete—the bridge initiated the deferred reply transaction, but now acts as its response agent; and the processor started out as the target and now serves as the request agent. In other words, the processor and the bridge have returned to the roles they originally played when the transaction was first initiated by the processor. Depending on how the transaction completed on the PCI side, the bridge delivers one of the following responses:

1. If the transaction completed successfully and the requested data was read from the target and stored in a temporary buffer in the bridge, the bridge must indicate the normal data response in the response phase of the deferred reply transaction.
2. If the transaction resulted in a target abort from the target, indicating that it is broken, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
3. If the transaction ended in a master abort because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed), the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.

Pentium Pro Processor System Architecture

4. If a read parity error was detected, the bridge may re-attempt the PCI transaction to see if the data can be read successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction). If it still fails or if no re-attempt is made, the bridge must indicate one of the following responses in the response phase of the deferred reply transaction:
 - the hard failure response, or
 - the normal data response, but pass the bad data and parity to the request agent as is.

Data Phase

Assuming that the read completed successfully on the PCI bus, the normal data response is issued by the bridge. The data phase of a read transaction starts when the normal data response is issued. The bridge asserts DBSY# to take ownership of the data bus. When it presents the data, it asserts DRDY#. In this case, it should be able to present the data quite rapidly because the data is being delivered from a temporary buffer within the bridge. When the last data item is being delivered, the bridge deasserts DBSY#, but keeps DRDY# asserted until the next clock. The processor samples the final data item and DRDY# asserted, indicating that it has latched valid data. The transaction has been completed.

Trackers Retire Transaction

The normal data response indicates that the data will be supplied to the request agent. Once the processor that initiated the original transaction as well as the other bus agents see the normal data response, the IOQs (i.e., the transaction trackers) in all bus agents retire the transaction from their deferred transaction queues.

Other Possible Responses

As indicated earlier, the PCI transaction may not have completed successfully. Table 14-2 on page 319 describes the relationship between the completion status of the PCI transaction and how the deferred reply transaction is completed.

Chapter 14: Transaction Deferral

Table 14-2: PCI Read Transaction Completion and Deferred Reply Completion

Completion Status of PCI Transaction	Completion Status of Deferred Reply Transaction
The transaction completed successfully and the requested data was read from the target and stored in a temporary buffer in the bridge.	The bridge must indicate the normal data response in the response phase of the deferred reply transaction and pass the read data back during the data phase.
The transaction resulted in a target abort from the target, indicating that it is broken.	The bridge must indicate the hard failure response in the response phase of the deferred reply transaction. There will be no data phase in the deferred reply transaction. When the processor receives the hard failure response, it generates a machine check exception.
The transaction ended in a master abort because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed)	The bridge must indicate the hard failure response in the response phase of the deferred reply transaction. There will be no data phase in the deferred reply transaction. When the processor receives the hard failure response, it generates a machine check exception.

Pentium Pro Processor System Architecture

Table 14-2: PCI Read Transaction Completion and Deferred Reply Completion (Continued)

Completion Status of PCI Transaction	Completion Status of Deferred Reply Transaction
PCI transaction received a retry.	<p>In this case, the PCI specification requires the host/PCI bridge to continually retry the PCI transaction until it completes successfully or is terminated by a target or master abort.</p> <p>It should be noted, however, that if the other bus were another Pentium Pro bus rather than a PCI bus and a retry response were received, the bridge is required to assert DEFER# in the snoop phase of the deferred reply transaction and must issue a retry response in the response phase. All bus agents then delete the transaction from their IOQs and the processor is then required to retry the transaction from scratch.</p>
A read parity error was detected. The bridge may re-attempt the PCI transaction to see if the data can be read successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction).	<p>If it still fails or if no re-attempt is made, the bridge must indicate one of the following responses in the response phase of the deferred reply transaction:</p> <ul style="list-style-type: none">• the hard failure response (when the processor receives the hard failure response, it generates a machine check exception), or• the normal data response, but pass the bad data and parity to the request agent as is. In this case, it would be left up to the processor to detect the corrupted data.

An Example Write

The previous section, "An Example Read" on page 311, described the actions of the bridge upon receipt of a read transaction that targeted a device residing on the PCI bus. This section describes the same scenario, but replaces the read with a write transaction.

Chapter 14: Transaction Deferral

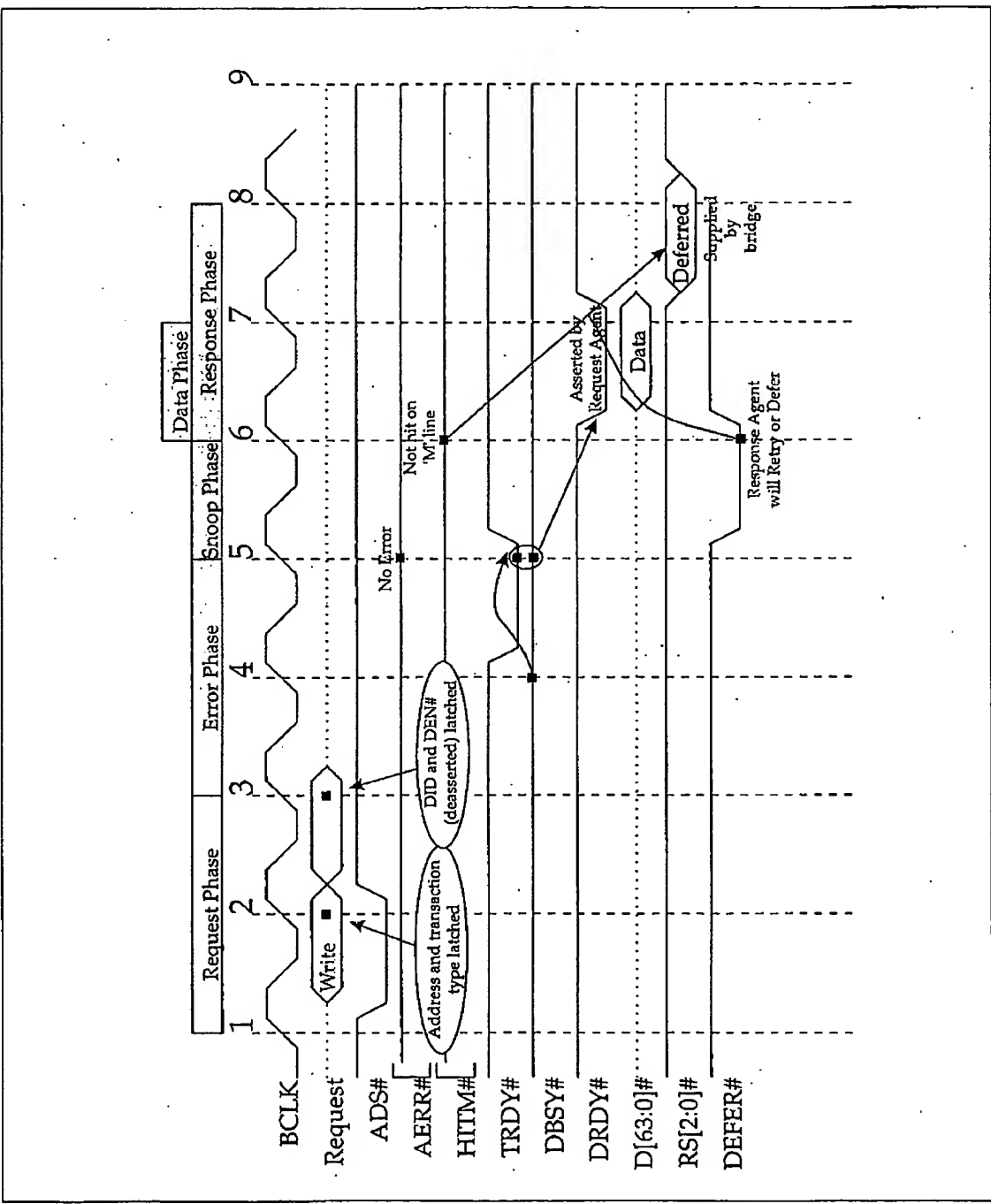
Transaction and Write Data Memorized, Deferred Response Issued

Refer to Figure 14-1 on page 308 and Figure 14-4 on page 322 during this example. Assume that a processor initiates a write transaction that targets a device residing beyond one of the host/PCI bridges:

1. The bridge memorizes the transaction, including the Deferred ID, or DID, delivered in request packet B.
2. This example assumes that the request initiator permits the response agent to defer the transaction (DEN# is asserted in request packet B).
3. The bridge asserts TRDY#, indicating its willingness to accept the write data into a buffer within the bridge.
4. When the request agent samples TRDY# asserted and DBSY# deasserted (the data bus is no longer in use by the previous owner), it takes ownership of the data bus one clock later (by asserting DBSY#). It then drives the data (one or more quadwords) to the bridge, asserting DRDY# as each is made available. The bridge latches the data.
5. In the snoop phase, the bridge asserts DEFER# to the request agent, indicating that it intends to issue a retry or a deferred response in the response phase.
6. In the response phase of the transaction, the bridge acts as the response agent and issues the deferred response. This causes the request agent to move the transaction from its IOQ into its deferred transaction queue. All other agents delete it from their IOQs. Effectively, the write transaction is now in a state of suspension until, at a later time, the response agent (i.e., the bridge) addresses the request agent using a deferred reply transaction.
7. During the deferred reply transaction that will be initiated by the bridge at a later time (when it has delivered the write data), the bridge will indicate the delivery status of the write data.

Pentium Pro Processor System Architecture

Figure 14-4: Write Transaction Receives Deferred Response



Chapter 14: Transaction Deferral

PCI Transaction Performed and Data Delivered to Target

As discussed earlier, the bridge performs the equivalent PCI write transaction to deliver the write data to the target device. The PCI transaction completes in one of the following ways:

1. the transaction **completes successfully**. The bridge notes the successful delivery of the data. In this case, the bridge must indicate the no data response in the response phase of the deferred reply transaction.
2. the transaction results in a **target abort** from the target, indicating that it is broken. In this case, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
3. the transaction ends in a **master abort** because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed). In this case, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
4. **Write parity error detected**. In this case, the bridge may re-attempt the PCI transaction to see if the data can be written successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction). If it still fails or if no re-attempt is made, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.

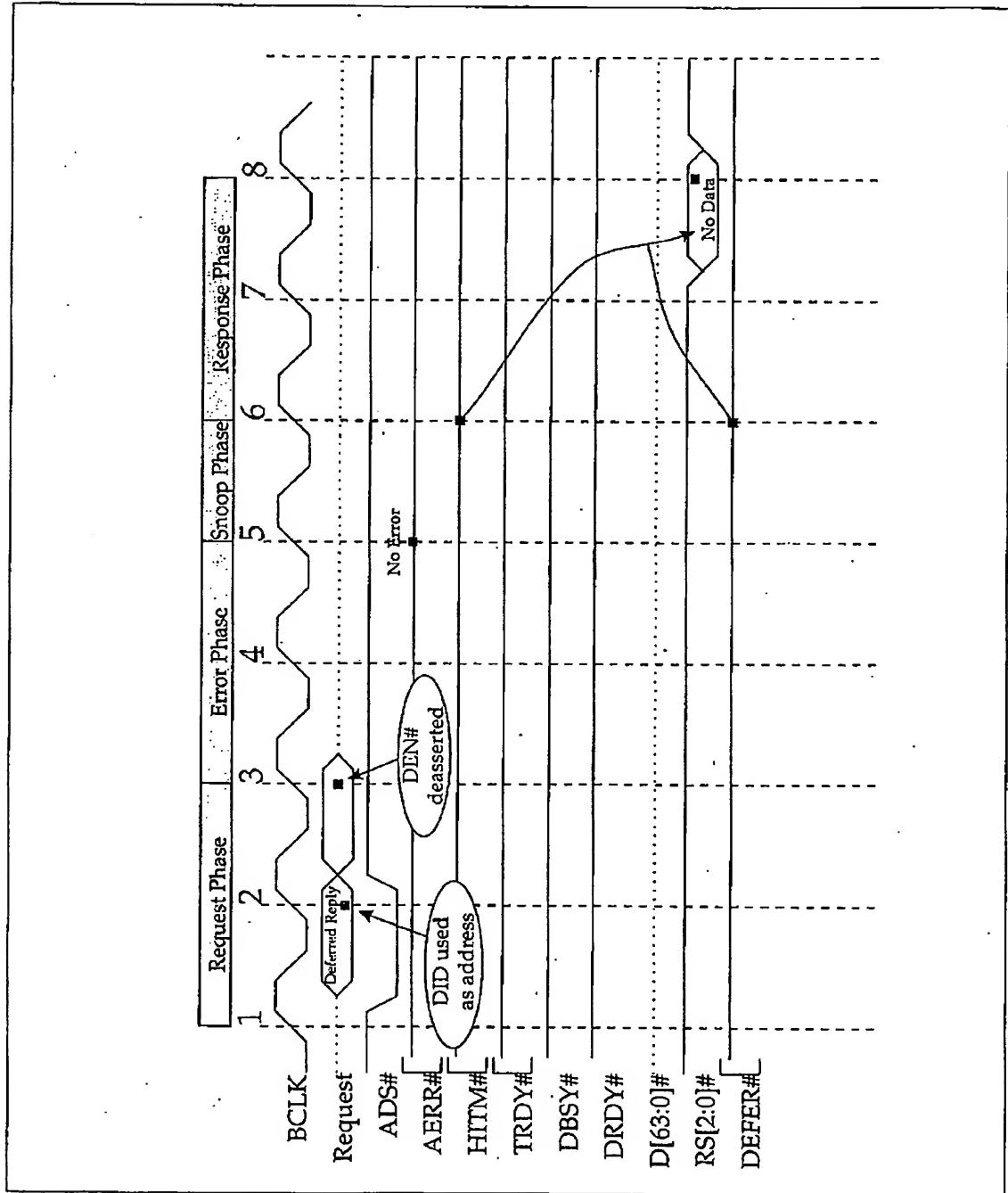
Deferred Reply Transaction Issued

Refer to Figure 14-5 on page 324 during the following discussion. When the PCI transaction has completed, the bridge uses BPRI# to arbitrate for ownership of the processor bus. Be aware that, in Figure 14-1 on page 308, the two host/PCI bridges both use the BPRI# signal to request ownership of the processor bus, but only one is permitted to drive it at a time. Before asserting BPRI#, therefore, if both of the bridges require ownership of the processor bus, they must use sideband signals to arbitrate between themselves for ownership of the BPRI# signal. The winner then asserts BPRI# to gain ownership of the request signal group.

Having acquired ownership of the request signal group, the bridge then issues a deferred reply transaction. During the request and error phases, the bridge acts as the request agent and the processor addressed by the deferred ID (see Table 14-1 on page 316) acts as the target of the transaction. The information indicated in Table 14-1 on page 316 is driven out during the transmission of packets A and B. All agents on the bus latch the two packets during the request phase, check parity, and issue AERR# in the error phase if there's a problem. If there isn't a problem, the transaction proceeds to the snoop phase.

Pentium Pro Processor System Architecture

Figure 14-5: Deferred Reply Transaction for Write



Chapter 14: Transaction Deferral

Original Request Agent Selected

All of the agents with outstanding deferred transactions compare the DID field latched in packet A (from A[23:16]#) to identify the previously-deferred transaction that has been in a state of suspension in their deferred transaction queue. The request agent that originated the write has a match on its agent type, agent ID, and transaction ID.

Bridge Provides Snoop Result

Until the snoop phase, the bridge is the request agent of the deferred reply transaction, but an interesting thing happens in the snoop phase. The deferred reply transaction is never snooped by the processor bus snoop agents. Rather, the snoop result is supplied by the bridge (i.e., the agent that deferred the transaction earlier) with the snoop result obtained from caches on the PCI bus when it did the write. Since typical system designs do not permit PCI agents to cache memory information, the snoop result delivered on HIT# and HITM# indicates a cache miss (both are deasserted). In addition, the DEFER# signal is not asserted (DEN# is always deasserted in request packet B during a deferred reply transaction).

Response Phase—Role Reversal

Once the snoop result is delivered to the processor by the bridge, the response phase of the transaction is entered. The role reversal is now complete—the bridge initiated the deferred reply transaction, but now acts as its response agent; and the processor started out as the target and now serves as the request agent. In other words, the processor and the bridge have returned to the roles they originally played when the transaction was first initiated by the processor. Depending on how the transaction completed on the PCI side, the bridge delivers one of the following responses:

1. If the transaction completed successfully and the data was written to the target, the bridge must indicate the no data response in the response phase of the deferred reply transaction.
2. If the transaction resulted in a target abort from the target, indicating that it is broken, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.
3. If the transaction ended in a master abort because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed), the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.

Pentium Pro Processor System Architecture

4. If a write parity error was detected, the bridge may re-attempt the PCI transaction to see if the data can be written successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction). If it still fails or if no re-attempt is made, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.

There is No Data Phase

The write data was transferred to the bridge in the data phase of the original write transaction that was deferred. The deferred reply transaction therefore completes with the response phase.

Trackers Retire Transaction

The no data response indicates that the data was accepted by the PCI target. Once the processor that initiated the original transaction as well as the other bus agents see the no data response, they retire the transaction from their deferred transaction queue.

Other Possible Responses

As indicated earlier, the PCI transaction may not have completed successfully. Table 14-3 on page 326 describes the relationship between the completion status of the PCI transaction and how the deferred reply transaction is completed.

Table 14-3: PCI Write Transaction Completion and Deferred Reply Completion

Completion Status of PCI Transaction	Completion Status of Deferred Reply Transaction
The transaction completed successfully and the write data was written to the target.	The bridge must indicate the no data response in the response phase of the deferred reply transaction.
The transaction resulted in a target abort from the target, indicating that it is broken.	The bridge must indicate the hard failure response in the response phase of the deferred reply transaction. When the processor receives the hard failure response, it generates a machine check exception.

Chapter 14: Transaction Deferral

Table 14-3: PCI Write Transaction Completion and Deferred Reply Completion (Continued)

Completion Status of PCI Transaction	Completion Status of Deferred Reply Transaction
The transaction ended in a master abort because no target responded to the transaction (i.e., DEVSEL# was not sampled asserted within the four PCI clocks after the transaction's address phase completed)	The bridge must indicate the hard failure response in the response phase of the deferred reply transaction. When the processor receives the hard failure response, it generates a machine check exception.
PCI transaction received a retry .	In this case, the PCI specification requires the host/PCI bridge to continually retry the PCI transaction until it completes successfully or is terminated by a target or master abort. <i>It should be noted, however, that if the other bus were another Pentium Pro bus rather than a PCI bus and a retry response were received, the bridge is required to assert DEFER# in the snoop phase of the deferred reply transaction and must issue a retry response in the response phase. All bus agents then delete the transaction from their IOQs and the processor is then required to retry the transaction from scratch.</i>
A write parity error was detected. The bridge may re-attempt the PCI transaction to see if the data can be written successfully on a second attempt (if the bridge has specific knowledge that the addressed device will not suffer side effects from a re-attempt of the transaction).	If it still fails or if no re-attempt is made, the bridge must indicate the hard failure response in the response phase of the deferred reply transaction.

Pentium Pro Support for Transaction Deferral

The Pentium Pro processor can have up to four of its transactions in the deferred (i.e., suspended) state at a given moment in time.

15

IO Transactions

The Previous Chapter

The previous chapter provided a detailed description of transaction deferral and the deferred reply transaction.

This Chapter

This chapter describes the specifics of IO read and write transactions from two angles:

- IO transactions as performed by the Pentium Pro processor.
- IO transactions that can be performed by agents other than the processor (or as they may be performed by future processors).

The Next Chapter

The next chapter provides a detailed description of the following transaction types:

- Interrupt acknowledge transaction.
- Special transaction.
- Branch trace message transaction.

Introduction

There is nothing exotic about IO transactions. Like any other transaction type, an IO transaction consists of a request, error, snoop, response and data phase. The following is a summary of general IO transaction characteristics:

- Since the processors never cache information from IO space, there will never be a hit on a cache line.
- The only appropriate snoop results are clean (HIT# and HITM# both deasserted), or snoop stall (both asserted).

Pentium Pro Processor System Architecture

- DEFER# may be asserted by the response agent if it intends to issue a retry or a deferred response in the response phase.
- In the response phase, the only response that may not be issued is the implicit writeback response (because there will never be a hit on a modified IO cache line).

IO Address Range

The IO address range supported by the Pentium Pro processor is from 000000000h through 000010002h (range is 64KB+3 in size). This is backward compatible with previous x86 processors (until reading this in the Pentium Pro data book, the author wasn't aware they did this). Consider the following:

- a 2-byte IO access starting at IO address FFFFh. In this case, the 2-bytes of data straddles the 64KB address boundary. Since these two bytes reside in different quadwords, the processor would perform this as two, separate, single-quadword transactions.
- a 4-byte IO access starting at IO address FFFFh, FFFEh, or FFFDh. As before, the target dword straddles the 64KB address boundary, and the processor would perform this as two, separate, single-quadword transactions.

In both cases, when accessing above the 64KB boundary, the processor would be asserting A[16]#.

Data Transfer Length

Behavior Permitted by Specification

When an IO read or write transaction is initiated, the data transfer length is output by the request agent in request packet B (see Table 11-4 on page 247). The specification permits IO data transfer lengths of:

- a quadword or less. Any combination of byte enables are valid, including none.
- two full quadwords. All byte enables must be asserted in request packet B.
- four full quadwords. All byte enables must be asserted in request packet B.

On a 0-byte read, the response must be the no data response, unless DEFER# is asserted by the response agent (indicating that it intends to retry or defer the transaction).

Chapter 15: IO Transactions

On a 0-byte write, the response agent must assert TRDY#, but the request agent must not assert DBSY# or DRDY# in response. Note that the author doesn't know why an agent would initiate a 0-byte IO transaction. The x86 processors are incapable of doing this.

How Pentium Pro Processor Operates

The Pentium Pro processor is only capable of initiating IO read and write transactions due to the execution of IO read (IN or INS) or write (OUT or OUTS) instructions. The programmer may only specify the AL, AX, or EAX register as the target or source register for the read or write. This restricts the transfers to:

- a single byte.
- two contiguous bytes.
- four contiguous bytes.

This means that, at most, the transfer length will always be less than a quadword and, at a maximum, four contiguous byte enables will be asserted. If the accessed data crosses a dword address boundary, the processor will behave as follows:

- if the transaction is an IO read and the access crosses the dword boundary within a quadword, one access is performed with the appropriate byte enables asserted.
- if the transaction is an IO read and the access crosses a quadword boundary, two separate single-quadword accesses are performed with the appropriate byte enables asserted.
- if the transaction is an IO write and the access crosses the dword boundary within a quadword, two accesses are performed with the appropriate byte enables asserted.
- if the transaction is an IO write and the access crosses a quadword boundary, two separate single-quadword accesses are performed with the appropriate byte enables asserted.

16

Central Agent Transactions

The Previous Chapter

The previous chapter described the specifics of IO read and write transactions from two angles:

- IO transactions as performed by the Pentium Pro processor.
- IO transactions that can be performed by agents other than the processor (or as they may be performed by future processors).

This Chapter

This chapter provides a detailed description of the following transaction types:

- Interrupt acknowledge transaction.
- Special transaction.
- Branch trace message transaction.

The Next Chapter

The next chapter focuses on the processor's remaining signals and concludes the description of the processor's hardware operation.

Point-to-Point vs. Broadcast

Most transactions are point-to-point transactions—the request agent addresses a specific area of memory or IO space for a read or a write and the addressed target acts as the transaction's response agent.

Some transactions generated by the processor don't target any specific memory or IO device, however. Rather, the processor is performing one of the following operations:

Pentium Pro Processor System Architecture

1. performing an **interrupt acknowledge transaction** to request the interrupt vector from the interrupt controller. In this case, the host/PCI bridge would act as the response agent (because the interrupt controller resides beyond the bridge).
2. performing a **special transaction** to broadcast a message to everyone. No one is targeted by the transaction, but someone has to act as the response agent. It is typically the host/PCI bridge.
3. performing a **branch trace message transaction** to inform a debug tool that, when executed, a branch was taken. Once again, no one in particular is being addressed and yet someone has to act as the response agent. It is typically the host/PCI bridge.

Intel refers to these as *central agent transactions* because one device typically acts as the default response agent for these transaction types. In a typical system (see Figure 16-1 on page 336), that device is typically the one Intel refers to as the compatibility bridge.

Interrupt Acknowledge Transaction

Background

An x86-based system usually incorporates an interrupt controller that receives interrupt requests from IO devices and passes them onto the processor (or processor cluster). The interrupt controller will either consist of a pair of 8259As in a single processor system, or an IO APIC module in a multi-processor system. Refer to Figure 16-1 on page 336. In Intel chipsets, the interrupt controller is incorporated in the PCI-to-ISA or PCI-to-EISA bridge. This is a strategically convenient place for it because the interrupt requests from PCI and EISA or ISA targets can easily be connected to it. As an example, the Intel ISA bridge (i.e., the SIO.A) incorporates the 8259A interrupt controllers (systems with ISA are typically single-processor systems), while their EISA bridge incorporates an IO APIC module (EISA systems are typically multi-processor systems targeting the server or workstation market).

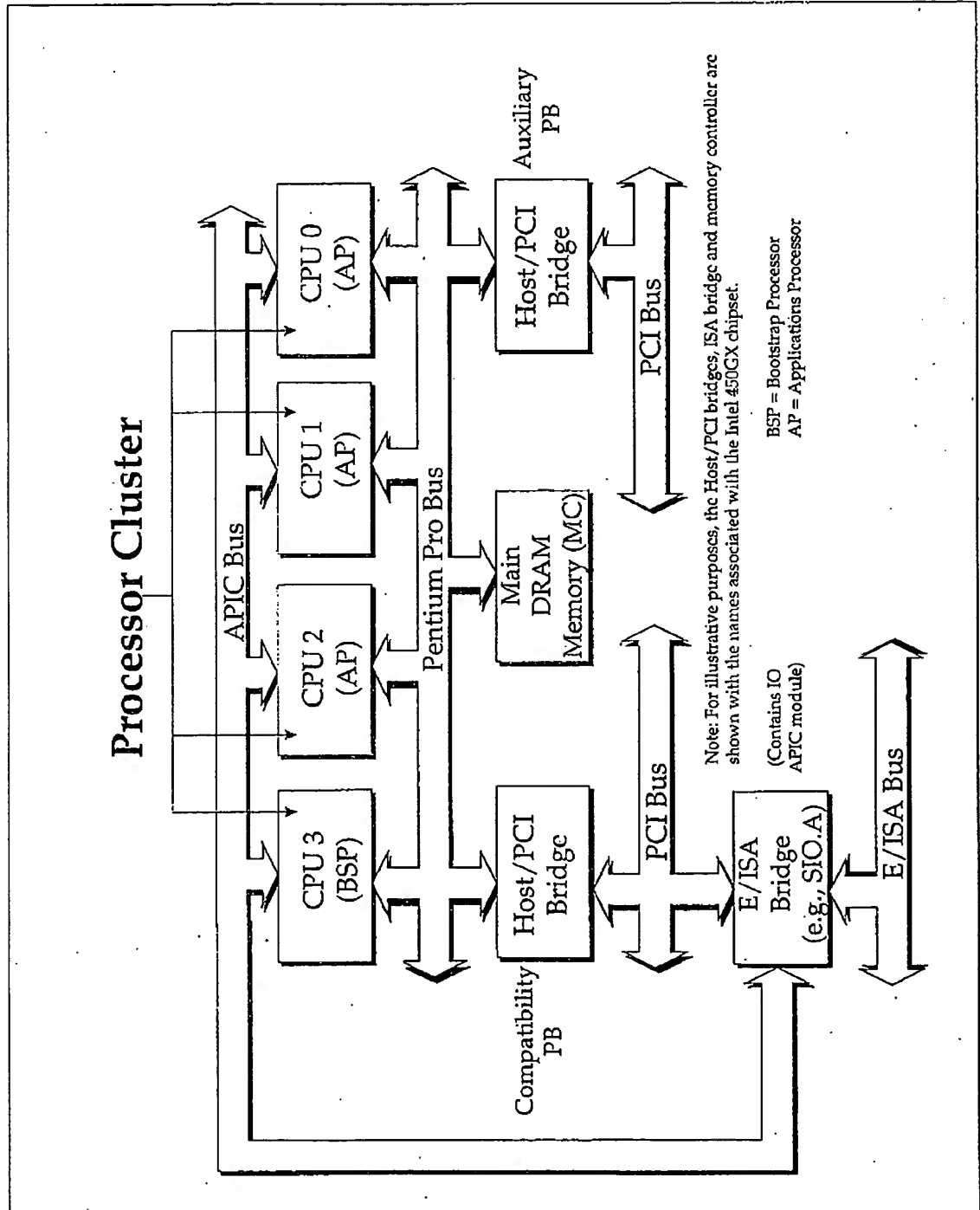
Assuming that the system uses the 8259A interrupt controller, the interrupt controller asserts its INTR (interrupt request) output when it detects any interrupt requests from IO devices. This is connected to the INTR pin (also referred to as the LINT0 pin) on one of the processors (typically the only processor). In response to its assertion, the processor takes the following actions:

Chapter 16: Central Agent Transactions

1. Assuming that recognition of external interrupts is enabled (in other words, the programmer has not executed a CLI instruction), the processor will recognize the request when it completes the execution of the current instruction.
2. The processor ceases to execute the interrupted program and pushes the contents of its CS, IP and EFLAGS registers into stack memory (to mark its place in the interrupted program).
3. The processor then disables recognition of additional external interrupts (i.e., it clears the IF bit in the EFLAGS register).

Pentium Pro Processor System Architecture

Figure 16-1: Typical System Block Diagram



Chapter 16: Central Agent Transactions

How Pentium Pro is Different

This is where the Pentium Pro processor is different than the previous x86 processors. The earlier processors generated two, back-to-back interrupt acknowledge transactions—one to command the interrupt controller to prioritize its pending requests, the second to request the interrupt vector for the most important one. However, the Pentium Pro processor only generates one interrupt acknowledge transaction. This transaction has the following characteristics:

- In packet A, the request type issued on REQ[4:0]# is 01000b (this is the logical, not electrical, value). For more information, refer to Table 11-4 on page 247.
- Although the content of the address bus in packet A is “don’t care,” it is factored into the address parity on AD[1:0]#.
- In packet B, REQ[4:0]# is 00x00b, where x is “don’t care.”
- In packet B, with the exception of A[15:8]# (the byte enables) and A[4]# (DEN#, defer enable), the content of the address bus is “don’t care.”
- In packet B, DEN# is asserted, granting the response agent permission to defer or retry the transaction if it so chooses.
- In packet B, only BE[0]# is asserted, indicating that it’s a single byte read to obtain the interrupt vector over data path 0 (D[7:0]#).

Host/PCI Bridge is Response Agent

In Figure 16-1 on page 336, the compatibility bridge acts as the response agent because the interrupt controller resides beyond the bridge (within the PCI/ISA bridge). Since it will take some time to obtain the vector from the other bridge, the host/PCI bridge may choose to issue a deferred response to the processor. The compatibility bridge arbitrates for ownership of the PCI bus and generates a PCI interrupt acknowledge transaction (see the chapter on commands in MindShare’s *PCI System Architecture* book, published by Addison-Wesley). When the PCI/ISA bridge latches the interrupt acknowledge transaction, it acts as the target of the transaction and provides the interrupt vector over PCI data path 0. The host/PCI bridge latches the vector and supplies it to the processor over data path 0. If the transaction had been deferred, a deferred reply transaction is used to provide the vector to the processor.

Pentium Pro Processor System Architecture

Special Transaction

General

Under special circumstances, an x86 processor generates a special transaction to broadcast a message to everyone. In other words, this is not a point-to-point transaction that targets a particular agent. As indicated earlier, every transaction requires that a response agent respond, and the compatibility bridge (see Figure 16-1 on page 336) is typically the response agent (i.e., the central agent) for the special transaction. This transaction has the following characteristics:

- Although the content of the address bus in packet A is "don't care," it is factored into the address parity on AD[1:0]#. For more information, refer to Table 11-4 on page 247.
- In packet A, the request type issued on REQ[4:0]# is 01000b (this is the logical, not electrical, value).
- In packet B, REQ[4:0]# is 00x01b, where x is "don't care."
- Although the content of A[35:16]# and A[7:3]# in packet B are "don't care," it is factored into the address parity on AD[1:0]#.
- In packet B, the byte enables indicate the type of message being broadcast (see next section).

Message Types

As stated in the previous section, the message type is driven out on BE[7:0]# (A[15:8]#) in packet B. Table 16-1 on page 339 indicates the types of messages that are currently defined.

Chapter 16: Central Agent Transactions

Table 16-1: Message Types

BE[7:0]#	Message Type
00h	Nop. Intel does not define what type of internal event causes the processor to generate this message.
01h	Shutdown. Indicates that the processor has incurred a severe software error. As with previous x86 processors, the Pentium Pro generates this message when it encounters a triple-fault condition. In other words, it has received another exception while attempting to call the double-fault exception handler. In response to the triple-fault, the processor ceases program execution and generates this message. Whether or not a system pays any attention to this message and, if so, the action taken by the system, is system design-specific. In a PC system, the host/PCI bridge generates a PCI special cycle transaction and broadcasts the shutdown message in the data phase of the PCI transaction. When detected by the PCI/ISA or PCI/EISA bridge, the bridge asserts reset to the system and then removes it. This causes the system to start over with the POST.
02h	Flush. Generated by the processor when it executes an INVD (invalidate caches) instruction. This instruction causes the processor to invalidate all of its internal caches without writing modified lines back to memory. The processor also broadcasts this message to inform any external caches that they should also invalidate their contents. It should be noted that this message does not cause other processors to invalidate their internal caches. If this is the programmer's intent, the processor's local APIC should be instructed to send an IPI (inter-processor interrupt message packet) to the other processors over the APIC bus that instructs them to do the same.
03h	Halt. Generated by the processor when it executes a HALT instruction. Whether or not a system pays any attention to this message and, if so, the action taken by the system, is system design-specific. In a PC system, the host/PCI bridge generates a PCI special cycle transaction and broadcasts the halt message in the data phase of the PCI transaction (in case any of the PCI agent care about the fact that a processor has halted).
04h	Sync. Generated by the processor when it executes an WBINVD (write back and invalidate caches) instruction. This instruction causes the processor to first write back all modified lines to memory, after which it invalidates all of its internal caches. The processor also broadcasts this message to inform any external caches that they should do the same. It should be noted that this message does not cause other processors to take this action in their internal caches. If this is the programmer's intent, the processor's local APIC should be instructed to send an IPI (inter-processor interrupt message packet) to the other processors over the APIC bus that instructs them to do the same.
05h	Flush Acknowledge. When external logic asserts the processor's FLUSH# input, this causes the processor to first write back all modified lines to memory, after which it invalidates all of its internal caches. The processor also broadcasts this message to inform system logic that it has completed the operation.

Pentium Pro Processor System Architecture

Table 16-1: Message Types (Continued)

BE[7:0]#	Message Type
06h	<p>Stop Grant Acknowledge. When external logic asserts the processor's STPCLK# (Stop Clock) input, the processor turns off the clock to all of its internal units with the exception of its external bus interface, the Time Stamp Counter (TSC), and the local APIC. This is referred to as the stop grant state and greatly diminishes the processor's power consumption. The processor also sets the Low Power Enable bit in its EBL_CR_POWERON MSR register to indicate that it is in the low power state. In addition, this message is broadcast to inform the system that the Stop Clock request has been honored.</p>
07h	<p>SMI Acknowledge. In response to a System Management interrupt received on SMI# (or via the APIC bus), the processor takes the following steps:</p> <ul style="list-style-type: none"> • ceases to execute the currently-executing program. • using the special transaction, broadcasts the SMI Acknowledge message. Note that the SMMEM# signal is asserted in packet B of this transaction. This informs the central agent that until it indicates otherwise, system management memory is being addressed. In response, Intel chipsets generate SMIACK# (SMI Active) to the DRAM controller to inform it that the processor will now be accessing SMM, not regular memory. • processor performs a series of memory writes to SMM to dump the contents of its register set to SMM. The processor is taking a "snapshot" of its context (i.e., register set contents) at the point of interruption. • processor then fetches and executes the SMM handler routine. • the final instruction in the SMM handler is always the RSM (Resume) instruction. When executed, the processor performs a series of memory reads to reload its register set from the register dump area. • after reloading its register set, the processor generates a second SMI Acknowledge message, this time with SMMEM# deasserted, informing the central agent that it will no longer be addressing SMM. In an Intel chipset, this causes the compatibility bridge to deassert the SMIACK# signal to the DRAM controller. • the processor then resumes execution of the interrupted program.
08h-FFh	Reserved.

Branch Trace Message Transaction Used for Program Debug

What's the Problem?

Before processors had internal caches, every memory read and write was performed on the bus. With a bus analyzer, you could therefore see every instruction that was fetched from memory for decode and execution. You could see when a branch instruction was fetched and, when it was subsequently decoded

Chapter 16: Central Agent Transactions

and executed, you would see the processor alter its program flow when the processor begins to issue memory read requests from the branch target address. In other words, you had full visibility to watch your program being executed.

When a processor incorporates an internal cache, however, program execution tracing becomes a real problem. After it reads a program into the internal code cache, you lose that visibility. You don't know what's going on. Specifically, you can't tell when the processor executes a branch instruction and, as a result, alters its program flow to a different part of the program that may already be in the code cache.

What's the Solution?

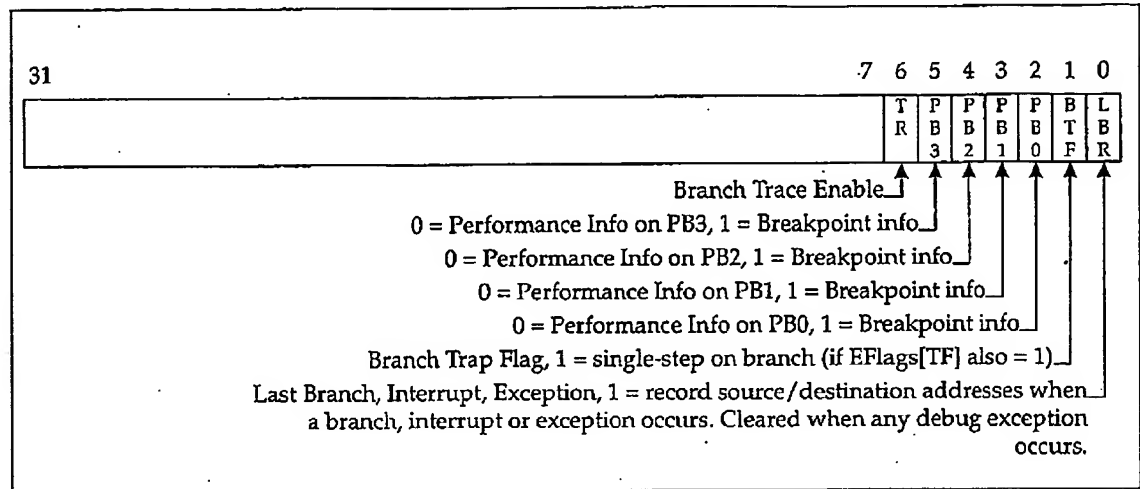
The Pentium and Pentium Pro processors can be forced to generate a message each time that it executes a branch and changes its program flow. The transaction type is the branch trace message transaction. This capability must be enabled by software, however.

Enabling Branch Trace Message Capability

The programmer enables or disables branch trace message transaction capability using a bit in an MSR (model-specific registers) called the `DEBUGCTL` (debug control) register (see Figure 16-2 on page 342). Once enabled, the processor generates a branch trace message transaction each time that a branch executes and causes a change in program flow (in other words, the branch to the new address is taken). This is the only central agent transaction that writes data. It writes information during the data phase to inform a debug tool which branch has been taken as well as the address that is being branched to.

Pentium Pro Processor System Architecture

Figure 16-2: DEBUGCTL MSR



Branch Trace Message Transaction

As stated earlier, interrupt acknowledge and branch trace message are the only central agent transactions that transfer data. Interrupt acknowledge reads the vector from the 8259A interrupt controller, and the branch trace message transaction writes data to the central agent.

Packet A Composition

A[35:3]# is reserved and can be any value. REQ[4:0]# contains 01001b. ADS# is asserted to indicate the transaction start. All three parity bits must be correct.

Packet B Composition

All three parity bits must be correct. REQ[4:0]# contains 00X00b, where the most-significant two bits are the DSZ field (see Table 11-4 on page 247) and most always be 00b as currently defined. 'X' means "don't care." A[35:16]# are reserved. A[15:8]#, the byte enables, are all asserted, indicating that eight bytes of data will be written in the data phase. A[7:3]#, the EXF signals, are set as follows:

- EXF4#, or A[7]#, is the SMMEM# (system management memory) signal and is deasserted.
- EXF3#, or A[6]#, is the SPLCK# (split lock) signal and is deasserted.

Chapter 16: Central Agent Transactions

- EXF2#, or A[5]#, is reserved and can be any value.
- EXF1#, or A[4]#, is the DEN# (defer enable) signal and is asserted. Intel doesn't overtly say this, but they do say that DEN# is always deasserted for locked transactions, deferred reply transactions, and 32 byte writes performed by the processor to cast a modified line back to memory to make room in the cache for a new line.
- EXF0#, or A[3]#, is reserved and can be any value.

Proper Response

The proper response is the no data response (because it's a write). Intel does not say that other responses are not permitted, so the author interprets this to mean that, although unlikely, the retry, deferred, and hard failure responses would also be permitted.

Data Composition

The central agent is responsible for asserting TRDY# to tell the processor to take ownership of the data bus and drive the data. The processor then drives out the eight bytes of data and asserts DRDY#. The author is assuming that it can do this with no wait states and therefore doesn't have to assert DBSY# (see "Special Case—Single Quadword, 0-Wait State Transfer" on page 301). When the data is driven, either the central agent or, more likely, a third party (the debug tool) snarfs the data. The data has the following composition:

- D[63:32]# contains the linear branch target address (i.e., the address branched to).
- D[31:0]# contains one of the following:
 - the linear start address of the branch instruction if its execution caused an exception (Intel uses the phrase "if the instruction does not complete normally").
 - the linear start address of the instruction immediately following the branch if the branch did not cause an exception.

17

Other Signals

The Previous Chapter

The previous chapter provided a detailed description of the following transaction types:

- Interrupt acknowledge transaction.
- Special transaction.
- Branch trace message transaction.

This Chapter

The previous chapters discussed all of the signals directly involved in performing bus transactions. This chapter focuses on the processor's remaining signals and concludes the description of the processor's hardware operation.

The Next Chapter

The next chapter begins the description of the processor's software characteristics.

Error Reporting Signals

In addition to the AERR# signal that is used in the error phase of the transaction, there are other error signals utilized by bus agents to signal various types of error conditions. The sections that follow describe these signals.

Bus Initialize (BINIT#)

Description

BINIT# is asserted by an agent when the bus cannot be reliably used for future transactions. As an example, if an agent's IOQ is corrupted, it can no longer reliably track transactions and therefore cannot reliably interact with the bus at the appropriate times. How a processor uses BINIT# is set up as follows:

Pentium Pro Processor System Architecture

- The ability of a processor to assert BINIT# can be configured at power-up time (in EBL_CR_POWERON MSR; see "Program-Accessible Startup Features" on page 45).
- A10# is sampled on the trailing-edge of reset to determine whether to observe BINIT# (see "Error Observation Options" on page 39).

When BINIT# is detected, all agents clear their IOQs, their deferred transactions queues, and also reset their bus interface state machines to the idle ownership state and rotating ID to 3.

The processor can be enabled to generate a machine check exception when BINIT# is detected or asserted. The machine check architecture registers can log an error indicating that either the processor observed someone else assert BINIT# or that it asserted BINIT#. The Intel documentation doesn't say, however, which processor will execute its machine check exception handler. The author assumes it would be the processor that asserted BINIT#. Multiple processors could observe it asserted by another agent and you typically wouldn't want all of them simultaneously executing the machine check exception handler.

There is the possibility that an agent other than a processor asserted BINIT#. In this event, the central agent (e.g., the compatibility bridge), could be configured to send an NMI to the processors over the APIC bus.

Assertion/Deassertion Protocol

See "BERR#/BINIT# Assertion/Deassertion Protocol" on page 347.

Bus Error (BERR#)

Description

BERR# is asserted to indicate that an error has been detected in the current transaction. An example would be an uncorrectable memory data error. Unlike BINIT#, however, BERR# assertion only applies to the current transaction, not to future transactions that may be generated on the bus. How a processor uses BERR# is set up as follows:

- A9# is sampled on the trailing-edge of reset to determine whether to observe BERR# (see "Error Observation Options" on page 39).
- Ability of a processor to assert BERR# is configured at power-up time (in EBL_CR_POWERON MSR (see "Program-Accessible Startup Features" on page 45).

Chapter 17: Other Signals

Using the machine check architecture MSRs, the processor can be enabled to generate a machine check exception on this event. Using the EBL_CR_POWERON MSR (see “Program-Accessible Startup Features” on page 45), the Pentium Pro processor can be configured to assert BERR#:

- to report an unrecoverable error (that is not handled by the machine check architecture) detected during a bus transaction when the processor is acting as the request agent.
- to report an internal error. In this case, BERR# is asserted once and then deasserted. In addition, IERR# is also asserted and remains asserted until NMI is recognized, or until RESET# or INIT# is asserted to the processor.

BERR#/BINIT# Assertion/Deassertion Protocol

BERR# and BINIT# are two of the six signals that may be driven by multiple agents simultaneously. The protocol demands that, when BERR# or BINIT# is asserted, it remain asserted for exactly three clocks.

- If an agent samples BERR# or BINIT# deasserted in the clock that it asserts it, it asserts BERR# or BINIT# for exactly three clocks and then releases it.
- If BERR# or BINIT# was already asserted by one or more other agents prior to an agent's assertion of BERR# or BINIT#, it must only assert BERR# or BINIT# for two clocks or one clock, depending on how long it has already been observed asserted. This results in BERR# or BINIT# never being asserted for more than three clocks.

Internal Error (IERR#)

IERR# is generated by a processor when an unrecoverable internal error is detected that is not handled by the machine check architecture logic (because it is disabled). The processor can also be configured to assert BERR# once along with IERR# (see the preceding section).

Functional Redundancy Check Error (FRCERR)

At startup time, a pair of processors can be set up as master and checker (see “FRC Mode Enable/Disable” on page 40). The checker watches the master and, if the master does something that the checker wouldn't have, the checker asserts FRCERR. The master can be configured (via its MSRs) to observe FRCERR and generates a machine check exception (if enabled via CR4[MCE]=1) on its assertion.

Pentium Pro Processor System Architecture

PC-Compatibility Signals

The processor's PC Compatibility signal group consists of FERR#, IGNNE#, and A20M#.

A20 Mask (A20M#)

A20M# allows the processor to emulate the address wrap-around at the 1MB address boundary that occurs on the 8086/8088 processors. This pin should only be asserted to the processor when the processor is operating in real mode. When A20M# is asserted to the processor, the processor masks physical address bit 20 (forces it to zero) before performing a snoop in the internal caches or driving a memory address onto the bus. A complete description and historical background can be found in the MindShare book entitled *ISA System Architecture* (published by Addison-Wesley).

FERR# and IGNNE#

For a detailed historical background on floating-point operation in the ISA world, refer to the MindShare book entitled *ISA System Architecture* (published by Addison-Wesley).

The programmer can select the processor's floating-point error handling methodology using the CR0[NE] bit. In an MS-DOS environment, the FP error handler is implemented as the IRQ13 handler and is incorporated within MS-DOS. This means that, when a FP error is encountered, the processor must somehow generate IRQ13 to the slave 8259A interrupt controller. The processor's FPU is embedded within the processor itself, so it must be configured to assert the processor's FERR# output when it encounters an error while attempting to execute a FP instruction. This is done by clearing CR0[NE] to zero. The processor then handles FP errors as follows:

- If external logic keeps the IGNNE# asserted, any FPU errors are ignored.
- If the IGNNE# input is deasserted and an unmasked FPU error is encountered while executing a FP instruction, the processor ceases program execution immediately before executing the next waiting FP instruction or WAIT/FWAIT instruction. The processor's FERR# output is asserted. Externally, FERR# is connected to the PC's IRQ13 interrupt request line. This causes the slave interrupt controller to generate a request to the master

Chapter 17: Other Signals

interrupt controller, which, in turn, generates INTR to the processor. The processor interrupts the currently-executing program and requests the interrupt vector from the interrupt controllers. The vector associated with IRQ13 (75h) is supplied to the processor and the processor jumps to the interrupt service routine pointed to by entry 75h in the interrupt table in memory. This is the MS-DOS FP error handler routine. The handler reads the FPU's status to determine the error condition. If the error is one that can be fixed by the handler, the handler then performs an IO write of all zeros to IO port F0h to clear the FPU's error condition. When the chipset sees this write, it asserts IGNNE# to the processor, permitting the FPU to proceed to the next instruction.

If CR0[NE] is set to one by the OS, the PC-compatible FPU error reporting method is not used. Instead, the processor generates an exception 16d when a FP error is encountered and the processor jumps directly to the FP error handler within the OS.

Diagnostic Support Signals

The Pentium Pro implements some pins specifically to aid in system and program debug. Those signals are described in Table 17-1 on page 349 and Table 17-2 on page 350.

Table 17-1: Diagnostic-Support Signals

Signal(s)	Description
BP[3:2]#	Breakpoint 3 and 2 output pins. Asserted by processor if a match is detected on either breakpoint 2 or 3. Starting with the 386, all x86 processors have incorporated a set of debug registers that can be used to set up and enable up to four program breakpoints (breakpoints 0 through 3). If an internal access matches the breakpoint conditions defined for one of these breakpoints, the processor asserts the respective BP output. For a detailed description of the debug registers, refer to the MindShare book entitled <i>Pentium Processor System Architecture</i> (published by Addison-Wesley). Also see the next entry.

Pentium Pro Processor System Architecture

Table 17-1: Diagnostic-Support Signals

Signal(s)	Description
BPM[1:0]#	Breakpoint outputs 0 and 1. These pins can be configured either as breakpoint indicators (as described in the preceding entry), or to alert a debug tool that either performance monitoring counter 0 or 1 has either overflowed or has been incremented. For more information refer to the chapter entitled "Performance Monitoring and Timestamp" on page 437.
TCK, TDI, TDO, TMS, TRST	Boundary Scan interface signals. Used for boundary scan testing and for connection to an external test tool (i.e., an ITP, or In-Target Probe).

Table 17-2: Probe Port Signals

Signal(s)	Description
PREQ#	Probe Request. Asserted by an ITP (in-target probe) tool to command the processor to enter probe mode (so the tool can access the processor's internal registers through the probe port (i.e., the boundary scan interface).
PRDY#	Probe Ready. Asserted by the processor in response to assertion of PREQ#. Indicates processor is in probe mode and ready to receive commands/requests via the boundary scan interface.

Interrupt-Related Signals

The processor's interrupt-related signals are described in Table 17-3 on page 350.

Table 17-3: Interrupt-Related Signals

Signal(s)	Description
RESET#	Hard reset. Clears all caches, returns processor to startup state. For a description of the startup state, refer to the chapter entitled "Processor Startup" on page 51.

Chapter 17: Other Signals

Table 17-3: Interrupt-Related Signals (Continued)

Signal(s)	Description
INIT#	Soft reset. Resets processor to startup state, but does not clear caches or affect floating-point registers. Also sampled on trailing-edge of reset to determine whether or not BIST should be run. This pin was first implemented on the 486 processor to avoid the massive performance dip when a 286 DOS extender program issues a reset to the processor to return the processor to real mode. For historical background, refer to the chapter entitled <i>The Reset Logic</i> in the MindShare book entitled <i>ISA System Architecture</i> (published by Addison-Wesley).
LINT0/INTR	Local Interrupt input 0 (in APIC mode), or INTR (Maskable External Interrupt) input (in 8259 mode). The local APIC can be configured to treat this pin either as a local interrupt pin or as the external interrupt line from the 8259A interrupt controller. For more information on LINT0, refer to the chapter entitled <i>The APIC</i> in the MindShare book entitled <i>Pentium Processor System Architecture</i> (published by Addison-Wesley). For more information on INTR, refer to the chapter entitled <i>The Interrupt Subsystem</i> in the MindShare book entitled <i>ISA System Architecture</i> (published by Addison-Wesley).
LINT1/NMI	Local Interrupt input 1 (in APIC mode), or Non-Maskable Interrupt input (in 8259 mode). The local APIC can be configured to treat this pin either as a local interrupt pin or as the NMI line from the PC chipset. For more information on LINT1, refer to the chapter entitled <i>The APIC</i> in the MindShare book entitled <i>Pentium Processor System Architecture</i> (published by Addison-Wesley). For more information on NMI, refer to the chapter entitled <i>The Interrupt Subsystem</i> in the MindShare book entitled <i>ISA System Architecture</i> (published by Addison-Wesley).
PICCLK	APIC clock line. Used to clock data between local APICs and IO APIC over PICD[1:0]. For more information, refer to the chapter entitled <i>The APIC</i> in the MindShare book entitled <i>Pentium Processor System Architecture</i> (published by Addison-Wesley).
PICD[1:0]	APIC data lines (see preceding entry).

Pentium Pro Processor System Architecture

Table 17-3: Interrupt-Related Signals (Continued)

Signal(s)	Description
SMI#	System Management Interrupt. Typically asserted by chipset when a power management event occurs. Causes processor to suspend normal operation and start execution of SMM handler. SMI can also be delivered over the APIC bus. For more information, refer to the chapter entitled <i>System Management Mode</i> in the MindShare book entitled <i>Pentium Processor System Architecture</i> (published by Addison-Wesley). Also refer to the description of the SMI Acknowledge message in Table 16-1 on page 339.

Processor Present Signals

The signals described in Table 17-4 on page 352 are used to indicate the presence of the processor and/or the upgrade processor.

Table 17-4: Processor Presence Signals

Signal	Description
UP#	Upgrade Present signal. Output from processor. In Pentium Pro, it's an open. In upgrade (Overdrive) processor, it's grounded. System board has pullup. Installing an Overdrive processor in lieu of the Pentium Pro asserts this signal. Used to disable voltage regulators that may be harmful to Overdrive processor.
CPUPRES#	CPU Present. Asserted by processor when installed.

Power Supply Pins

Table 17-5 on page 353 describes the processor's power supply-related pins.

Chapter 17: Other Signals

Table 17-5: Processor Power-related Pins

Signal	Description
POWERGOOD	3.3V-tolerant input. When asserted, indicates that clocks and the 3.3V, 5V, and VccP supplies are stable and within spec. When asserted, must be immediately followed by a minimum 1ms assertion of RESET#. Used to protect internal circuits from voltage sequencing issues.
Vcc5	5Vdc for processor cooling fan.
VccP	CPU die operating voltage.
VccS	L2 cache die operating voltage.
VID[3:0]	Voltage ID. Identifies CPU die's power supply requirements (see Table 17-6 on page 353).
VREF[7:0]	Reference voltage inputs for GTL+ receivers (see "Everything's Relative" on page 180). Nominally = 1.0Vdc.
Vss	Ground.

Table 17-6: Voltage ID Encoding

VID[3:0] Value	Operating Voltage Required by CPU Die
0000b	3.5
0001b	3.4
0010b	3.3
0011b	3.2
0100b	3.1
0101b	3.0
0110b	2.9
0111b	2.8
1000b	2.7

Pentium Pro Processor System Architecture

Table 17-6: Voltage ID Encoding (Continued)

VID[3:0] Value	Operating Voltage Required by CPU Die
1001b	2.6
1010b	2.5
1011b	2.4
1100b	2.3 (support not expected)
1101b	2.2 (support not expected)
1110b	2.1 (support not expected)
1111b	CPU not present

Miscellaneous Signals

Table 17-7 on page 354 describes the processor's remaining signals.

Table 17-7: Miscellaneous Signals

Signal(s)	Description
BCLK	Bus Clock input. 3.3V signal. Used directly as bus clock. Also multiplied internally to yield processor core's internal PCLK (see "Processor Core Speed Selection" on page 41).
FLUSH#	When asserted, causes processor to write back all modified lines to memory and invalidate all cache entries. Processor then performs a Flush Acknowledge special transaction (see "Special Transaction" on page 338) to indicate that flush is complete. Also sampled at trailing-edge of reset to determine if processor should tri-state all of its outputs (see "Selecting Tri-State Mode" on page 41)
PLL[2:1]	Phase-Locked Loop decoupling pins. Should be connected to an external decoupling capacitor.

Chapter 17: Other Signals

Table 17-7: Miscellaneous Signals (Continued)

Signal(s)	Description
STPCLK#	Stop Clock. When asserted to processor, causes processor to disable internal clock to all units except the bus interface, Timestamp Counter and APIC. Processor enters the Stop Grant state and generates a Stop Grant Acknowledge special transaction (see "Message Types" on page 338) to indicate that it has stopped the internal clock. Processor will still snoop external memory accesses by other agents.
TESTHI	Test High (3 pins). Must all be pulled up (preferred) or may be tied directly to VccP.
TESTLO	Test Low (13 pins). Must all be pulled low (preferred) or may be connected directly to Vss.
THERMTRIP#	When junction temperature exceeds ~ 135C, the processor stops execution, asserts this output. A reset will restart program execution if the junction temperature has fallen below the trip point. Otherwise, the processor remains stopped.

Part 3:

Processor's Software

Characteristics

The Previous Part

Part 2 of the book provided a detailed description the hardware environment.

This Part

Part 3 of the book provides a detailed description of the enhancements to the software enviroment that have been introduced in the Pentium Pro processor. This part is divided into the following chapters:

- "Instruction Set Enhancements" on page 359.
- "Register Set Enhancements" on page 373.
- "Paging Enhancements" on page 379.
- "Interrupt Enhancements" on page 401.
- "Machine Check Architecture" on page 415.
- "Performance Monitoring and Timestamp" on page 437.
- "MMX: Matrix Math Extensions" on page 445.

The Next Part

Part 4 of the book provides an overview of the Intel 450KX, 450GX, and 440FX chipsets.

18 *Instruction Set Enhancements*

This Chapter

This chapter describes new instructions as well as enhancement of previously-implemented instructions.

The Next Chapter

The next chapter describes new registers, the bits within them, and the new features that they are associated with.

Introduction

The purpose of this chapter is to introduce the new instructions that have been added to the Pentium Pro processor. For a complete description of each of these instructions, refer to the Intel Pentium Pro *Programmer's Reference Guide*.

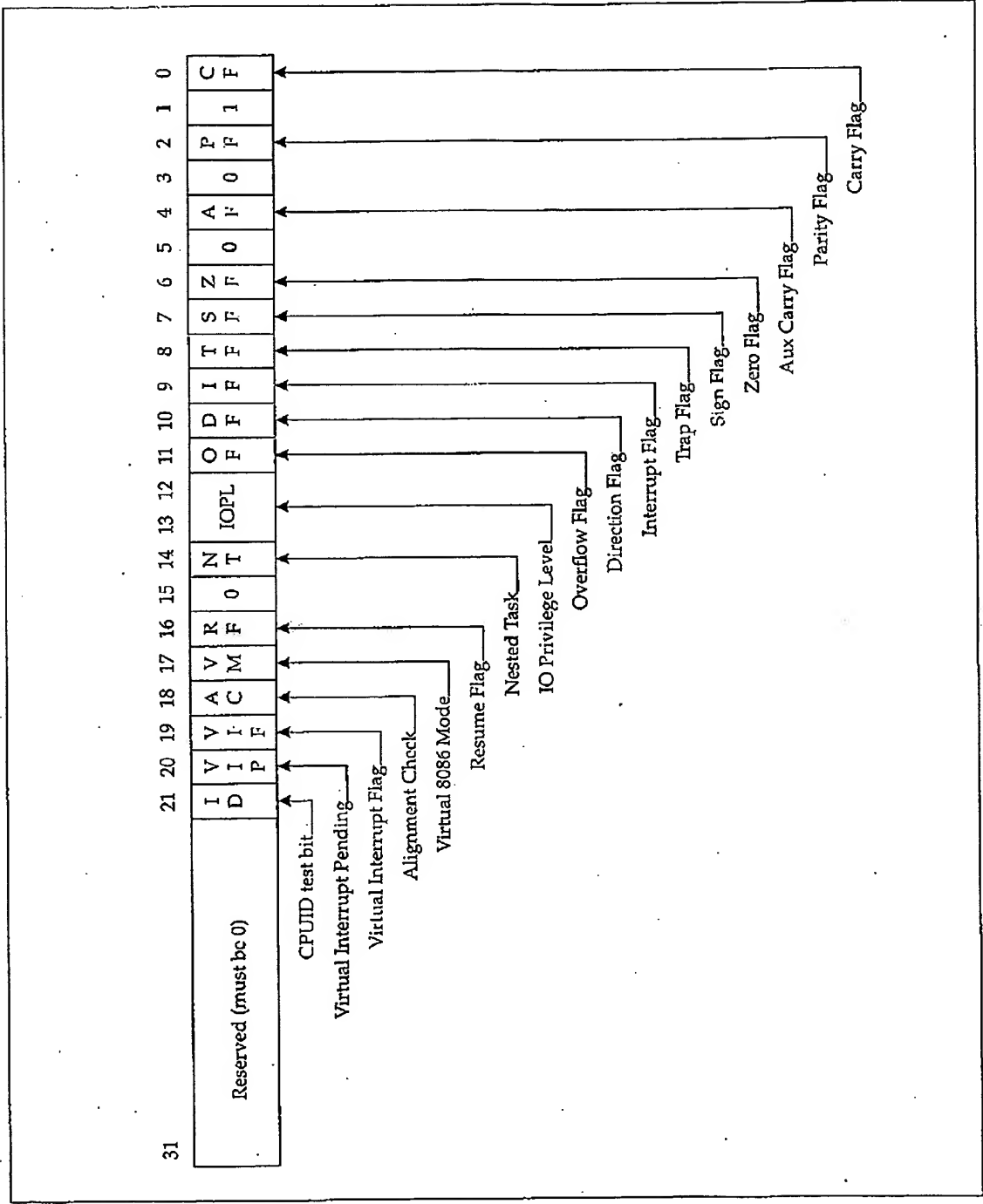
CPUID Instruction Enhanced

Before Executing, Determine if Supported

Before executing the CPUID instruction, the programmer must first ascertain if the processor implements it. This is accomplished by attempting to write a one into the EFLAGS[ID] bit (see Figure 18-1 on page 360). If the bit can be changed to a one, then the processor supports it.

Pentium Pro Processor System Architecture

Figure 18-1: EFLAGS Register



Chapter 18: Instruction Set Enhancements

Basic Description

When executed, the CUID instruction returns information about the processor in the processor's registers. The CUID instruction was first implemented in the first version of the Pentium processor. All subsequently-introduced versions of the Pentium and the 486 processors implemented CUID. However, prior to the Pentium Pro processor, the amount of information returned by the CUID instruction was minimal. It consisted of the vendor ID string and the processor family, model and stepping (i.e., the revision level of the processor silicon).

When executed by the Pentium Pro processor, the CUID instruction can return a wealth of information about the processor. This includes:

- processor family, model and stepping (i.e., revision).
- features supported by the processor.
- the size and structure of all of the processor's internal caches. This includes the L1 and L2 caches, as well as the data and code TLBs (Translation Lookaside Buffers).

Before executing the CUID instruction, the programmer first loads a request type indicator into the EAX register. When the CUID instruction is executed, the processor examines the value in EAX to determine what information is being requested. The requested information is then supplied in the processor's registers. Different processor implementations may support different request values. The EAX values defined for the current implementation of the processor are:

- EAX = 0 indicates a request for the vendor ID string and the maximum EAX input value (i.e., request type) supported by the processor.
- EAX = 1 indicates a request for the processor version information and the features supported by the processor.
- EAX = 2 indicates a request for information about the processor's caches and TLBs.

Vendor ID and Max Input Value Request

Before issuing other request types to the processor, the EAX input value of 0 should be used first to ascertain the request types supported by this processor implementation. As indicated earlier, in response to the type 0 request, the processor provides the maximum EAX value (i.e., request type) supported by the processor. Armed with this information, request for additional information may be issued.

Pentium Pro Processor System Architecture

Request for Vendor ID String and Max EAX Value

Request type 0 returns the processor's vendor ID string and the max-allowable EAX request type value. The vendor ID string is returned as packed ASCII data in the EBX, ECX and EDX registers. The maximum-allowable EAX input value (currently = 2) is returned in EAX.

Request for Version and Supported Features

Refer to Figure 18-2 on page 363. Request type 1 returns the processor version (in EAX) and the features it supports (in EDX). The processor types currently defined are listed in Table 18-1 on page 363. The Pentium Pro belongs to the sixth generation, or family, of x86 processors. The contents of the EDX register indicate the features supported by the processor. The current versions of the Pentium Pro processor return all ones in the currently-defined bits. Refer to the indicated sections for an explanation of each of these features:

- CMOV. Refer to "Conditional Move (CMOV) Eliminates Branches" on page 367.
- Machine check architecture. "Machine Check Architecture" on page 415.
- Global page feature. "Paging Enhancements" on page 379.
- MTRRs. "Rules of Conduct" on page 119.
- APIC. A detailed description the APIC can be found in the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).
- CXS. A description of the Compare and Exchange 8 Bytes instruction can be found in the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).
- Machine check exception. "Machine Check Architecture" on page 415.
- Physical address extension (PAE). "Paging Enhancements" on page 379.
- RDMSR and WRMSR. "Accessing MSRs" on page 371.
- Time stamp counter (TSC). "Time Stamp Counter Facility" on page 438.
- Page size extensions (PSE). "Paging Enhancements" on page 379.
- Debug extensions. A detailed description the debug extensions can be found in the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).
- VM86 mode extensions. "Interrupt Enhancements" on page 401.
- FPU present. The meaning of this bit speaks for itself.

Chapter 18: Instruction Set Enhancements

Figure 18-2: Processor Version and Features Supported Information

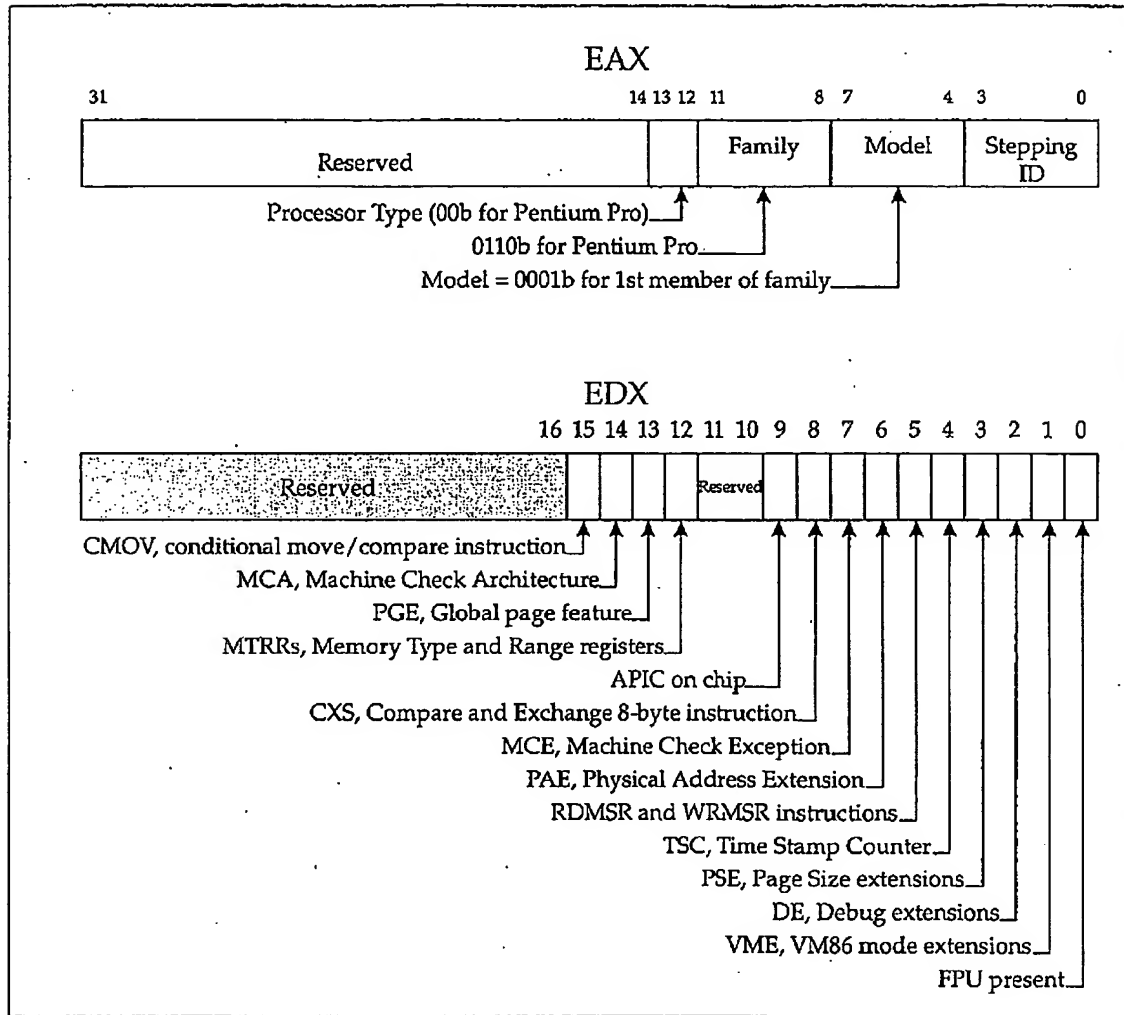


Table 18-1: Processor Type Field

Processor Type	Description
00b	Original OEM processor.
01b	Overdrive processor.
10b	Dual-processor.
11b	Reserved.

Pentium Pro Processor System Architecture

Request for Cache and TLB Information

Request type 2 returns information about the processor's caches and TLBs. Each processor has its own number of caches and TLBs and the size and architecture of each is processor design-dependent. This information is returned in the EAX, EBX, ECX and EDX registers. The information is formatted as a series of descriptors, each one byte in size. Note that these four registers may not be big enough to hold all of the descriptors necessary to describe a processor's caches and TLBs. In this case, the CUID instruction would have to be executed multiple times, each time using request type 2, to obtain all of the descriptors pertaining to the processor. Upon executing the first type 2 request, the value returned in the least-significant byte of the EAX register indicates how many times the CUID instruction must be executed with request type 2 to obtain all of the descriptors. Table 18-2 on page 364 indicates the descriptor types currently defined. Table 18-3 on page 365 defines the values returned for the current implementation of the Pentium Pro processor.

Table 18-2: Currently-Defined Cache/TLB Descriptors

Hex Value	Description
00h	Null descriptor.
01h	The TLB for page table entries related to 4KB code pages is 4-way set-associative with 64 entries.
02h	The TLB for page table entries related to 4MB code pages is 4-way set-associative with 4 entries.
03h	The TLB for page table entries related to 4KB data pages is 4-way set-associative with 64 entries.
04h	The TLB for page table entries related to 4MB data pages is 4-way set-associative with 8 entries.
06h	The L1 code cache is 8KB in size, organized as a 4-way set-associative with 32 bytes per line.
0Ah	The L1 data cache is 8KB in size, organized as a 2-way set-associative with 32 bytes per line.

Chapter 18: Instruction Set Enhancements

Table 18-2: Currently-Defined Cache/TLB Descriptors

Hex Value	Description
41h	The unified L2 cache is 128KB in size, organized as a 4-way set-associative cache with 32 bytes per line.
42h	The unified L2 cache is 256KB in size, organized as a 4-way set-associative cache with 32 bytes per line.
43h	The unified L2 cache is 512KB in size, organized as a 4-way set-associative cache with 32 bytes per line.

Table 18-3: Descriptors Returned by One of the Current Processor Implementations

Register	Byte	Value	Description
EAX	0	01h	One execution of CPUID with request type 2 returns all descriptors.
	1	01h	The TLB for page table entries related to 4KB code pages is 4-way set-associative with 64 entries.
	2	02h	The TLB for page table entries related to 4MB code pages is 4-way set-associative with 4 entries.
	3	03h	The TLB for page table entries related to 4KB data pages is 4-way set-associative with 64 entries.
EBX	0-3	00h	Null descriptors.
ECX	0-3	00h	Null descriptors.
EDX	0	42h	The unified L2 cache is 256KB in size, organized as a 4-way set-associative cache with 32 bytes per line.
	1	0Ah	The L1 data cache is 8KB in size, organized as a 2-way set-associative with 32 bytes per line.
	2	04h	The TLB for page table entries related to 4MB data pages is 4-way set-associative with 8 entries.
	3	06h	The L1 code cache is 8KB in size, organized as a 4-way set-associative with 32 bytes per line.

Pentium Pro Processor System Architecture

CPUID is a Serializing Instruction

As described in the chapter entitled "The Fetch, Decode, Execute Engine" on page 61, the processor executes instructions in the ROB out-of-order. This means that an instruction found later in a program's flow may be executed before instructions found earlier in that program's flow. Sometimes it is important that all instructions up to a certain point in program execution have completed execution before executing any instructions beyond that point. The following is an example where this would be true.

Assume that the programmer wishes to time how long a particular portion of a program takes to execute. Before executing the code to be timed, the programmer takes a snapshot of the current time by reading the current contents of the Time Stamp Counter, or TSC. The portion of code in question is then executed, after which the TSC is read again. In a processor that doesn't execute instructions out-of-order, you are guaranteed that the second read of the TSC will not be executed until all of the instructions in the code sequence have completed execution. However, if the same program is executed by a processor that performs out-of-order execution (such as the Pentium Pro), the second TSC read may be performed before all of the instructions that precede it in the program have actually been executed. The elapsed time reading is therefore invalid. In addition, instructions that follow the second TSC read may already have been executed.

This problem can be fixed by preceding the second read of the TSC with a serializing instruction (such as CPUID). When the processor detects a serializing instruction in the ROB, it will not execute any of the instructions that follow the serializing instruction until all of the instructions that precede it have completed execution and all changes to flags registers and memory (the processor's posted write buffers are flushed) have been accomplished. This insures that all work has been completed up to the serializing instruction before it completes its execution.

Only when these conditions have been met are any of the instructions that follow the serializing instruction executed. The CPUID instruction can be executed at any privilege level and in both real and protected mode.

Chapter 18: Instruction Set Enhancements

Serializing Instructions Impact Performance

Because serializing instructions prevent the processor from performing speculative execution from micro-ops that reside downstream within the ROB, they have a negative effect on performance. Liberal use of serializing instructions can noticeably impact performance, so they should only be used when absolutely necessary.

Conditional Move (CMOV) Eliminates Branches

As was stressed in “The Fetch, Decode, Execute Engine” on page 61, the processor has a deep instruction pipeline and also executes instructions out-of-order. For these reasons, mispredicted branch instructions can cause a fairly substantial decrease in performance. When a branch is executed and it is determined that its branch path was predicted incorrectly, all of the instructions currently in the prefetch streaming buffer and the earlier instruction pipeline stages must be flushed. In addition, any instructions that had been speculatively executed that occur after the branch in the program must also be deleted from the ROB.

The ideal program would have no branches, or only unconditional branches. Since this isn’t realistic, however, a better plan would be to limit, as much as possible, the number of conditional branches found in a program. The conditional move instruction (CMOV) permits the elimination of a conditional branch by testing a condition and only performing the indicated move if the condition tests true.

Conditional FP Move (FCMOV) Eliminates Branches

The floating-point conditional move (FCMOV) is the floating-point equivalent of the CMOV instruction. It conditionally moves values between floating-point stack registers and permits the elimination of a conditional branch. Prior to executing the FCMOV instruction, the programmer would execute an FCOMI, FCOMIP, FUCOMI, or FCOMIP instruction (see next section) to set the appropriate condition bits in the integer EFLAGS register to be tested by the FCMOV instruction.

Pentium Pro Processor System Architecture

FCOMI, FCOMIP, FUCOMI, and FUCOMIP

The following instructions have been added to the floating-point instructions set and are intended for use with the FCMOV instruction:

- **FCOMI.** FP compare real and set integer flags (in the EFLAGS register, rather than the FPU flags) instruction.
- **FCOMIP.** FP compare real and set integer flags instruction. Also pops the FP register stack.
- **FUCOMI.** FP unordered compare real and set integer flags instruction
- **FUCOMIP.** FP unordered compare real and set integer flags instruction. Also pops the FP register stack.

Read Performance Monitoring Counter (RDPMC)

Technically, the RDPMC instruction was introduced in the Pentium processor. However, its definition and usage weren't documented in the public-domain documentation. Rather, it was described in the infamous appendix H. Intel has chosen to reveal most of the appendix H information in the Pentium Pro documentation, so it is being described here.

What's RDPMC Used For?

The Pentium and Pentium Pro processors incorporate performance monitoring logic that can be set up to record the duration of or number of occurrences of a particular event. Both processors implement two counters, each of which can be set up to monitor for different types of events. A detailed description of the performance monitoring facility can be found in "Performance Monitoring and Timestamp" on page 437.

This instruction is used to obtain (i.e., read) the current contents of one of the performance monitoring counters.

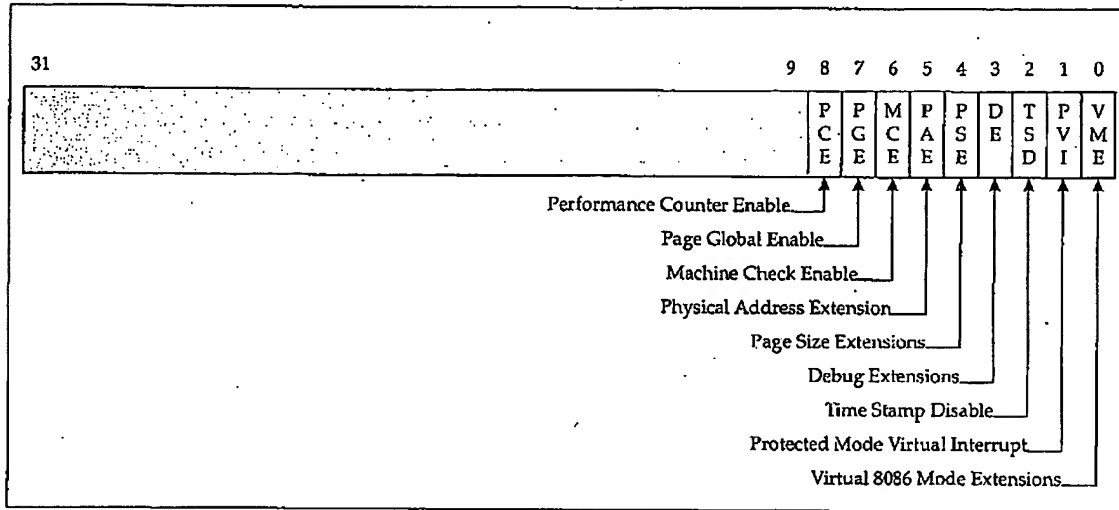
Who Can Execute RDPMC?

The CR4[PCE] bit governs the privilege level that the RDPMC instruction can be successfully executed at. When CR4[PCE] = 1 (see Figure 18-3 on page 369), programs executing at any privilege level can execute this instruction success-

Chapter 18: Instruction Set Enhancements

fully. When CR4[PCE] = 0, only kernel OS code executing at privilege level 0 can successfully execute this instruction.

Figure 18-3: CR4



RDPMC Not Serializing Instruction

The RDPMC instruction is not a serializing instruction, so it may not obtain an accurate event count or duration (see "CPUID is a Serializing Instruction" on page 366 for more information on the need for serializing instructions). To ensure that all instructions prior to the RDPMC have completed execution and their results have been committed to registers and memory, precede the RDPMC with a serializing instruction (such as CPUID).

RDPMC Description

Prior to executing the RDPMC instruction, the programmer first specifies which of the performance monitoring counters is to be read (the current versions of the Pentium and Pentium Pro processors only implement counters 0 and 1; future processors could implement more). The ECX register is loaded with the target counter number (e.g., 0 or 1). Upon execution, the contents of the target counter are placed in the EDX:EAX register pair, with the upper 8 bits being placed in the lower 8 bits of EDX (the current implementations of the counters are 40 bits each) and upper 24 bits of EDX are zeroed.

Pentium Pro Processor System Architecture

Read Time Stamp Counter (RDTSC)

Like the RDPMC instruction, the RDTSC instruction was introduced in the Pentium processor. However, its definition and usage weren't documented in the public-domain documentation. Rather, it was described in the infamous appendix H. Intel has chosen to reveal most of the appendix H information in the Pentium Pro documentation, so it is being described here.

What's RDTSC Used For

The Pentium and Pentium Pro processors incorporate a 64-bit time stamp counter (TSC) that is cleared to 0 at reset and then is incremented once for each processor clock cycle. A detailed description of the time stamp facility can be found in "Time Stamp Counter Facility" on page 438 and in "Performance Monitoring and Timestamp" on page 437. This instruction is used to obtain (i.e., read) the current contents of one of the TSC.

Who Can Execute RDTSC?

The CR4[TSD] bit (see Figure 18-3 on page 369) governs the privilege level that the RDTSC instruction can be successfully executed at. When CR4[TSD] = 0, programs executing at any privilege level can execute this instruction successfully. When CR4[TSD] = 1, only kernel OS code executing at privilege level 0 can successfully execute this instruction.

RDTSC Doesn't Serialize

The RDTSC instruction is not a serializing instruction, so it may not obtain an accurate count of the number of processor clock cycles that have elapsed while the previous instructions have executed (see "CPLID is a Serializing Instruction" on page 366 for more information on the need for serializing instructions). To ensure that all instructions prior to the RDTSC have completed execution and their results have been committed to registers and memory, precede the RDTSC with a serializing instruction (such as CPLID).

Chapter 18: Instruction Set Enhancements

RDTSC Description

Upon execution, the contents of the 64-bit TSC is placed in the EDX:EAX register pair, with the upper 32 bits placed in EDX. The TSC may also be read using the RDMSR instruction (see "Accessing MSRs" on page 371).

My Favorite—UD2

UD2 is my favorite instruction: it is *by definition an architecturally defined undefined instruction* that, when executed, is guaranteed to generate an undefined opcode exception in any of the processor's operating modes. It was included in the instruction set for testing purposes, permitting the deliberate generation of an undefined opcode exception on any processor.

Accessing MSRs

The Pentium processor was the first x86 processor to include model-specific registers, or MSRs. The RDMSR and WRMSR instructions were defined to permit the MSRs to be read and written. Although these two instructions were not introduced in the Pentium Pro processor, there are many more MSRs implemented than in the Pentium and, consequently, these two instructions now accept many more input parameters than in the Pentium.

Testing for Processor MSR Support

Before executing either of these instructions, use the CPUID instruction with request type one to get the processor version and features information. The features supported information is returned in the EDX register and EDX[5] will be set to one if the instructions are supported (see Figure 18-2 on page 363).

Causes GP Exception If...

These instructions can only be executed in real mode or when executing at privilege level 0 in protected mode. If these conditions are violated, the processor generates a GP exception.

Pentium Pro Processor System Architecture

Input Parameters

Before executing either instruction, the programmer first loads the MSR address into the ECX register (the MSR addresses are listed in appendix C of the *Intel Operating System Writer's Guide*). When executing the WRMSR instruction, the 64-bit data value to be written to the selected MSR must be loaded into the EDX:EAX register pair. When executing the RDMSR instruction, the contents of the selected MSR is returned in the EDX:EAX register pair.

19 *Register Set Enhancements*

The Previous Chapter

The previous chapter provided a description of new instructions as well as enhancement of previously-implemented instructions.

This Chapter

This chapter describes new registers, the bits within them, and the new features that they are associated with. It also describes new bits added to pre-existent registers and the features that they are associated with.

The Next Chapter

The next chapter describes the physical address extension (PAE) feature introduced in the Pentium Pro processor. This feature permits programs to access code or data within a 64GB rather than a 4GB range. In future processors, the PAE could permit access to up to 2^{64} memory space.

New Registers

Introduction

The following new registers were introduced in the Pentium Pro processor:

- **New local APIC LVT entry.** A description of this enhancement can be found in "Interrupt Enhancements" on page 401 and in "Performance Monitoring and Timestamp" on page 437. Please note that the performance monitoring features were added in the Pentium processor, but were only documented with the release of the Pentium Pro processor.

Pentium Pro Processor System Architecture

- **DebugCTL, LastBranch and LastException MSRs.** A description of this enhancement can be found in "DebugCTL, LastBranch and LastException MSRs" on page 374.
- **Time Stamp Counter (TSC) MSR.** A description of this enhancement can be found in "Time Stamp Counter Facility" on page 438. Please note that the Time Stamp Counter feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **MTRRs.** A description of this enhancement can be found in "Rules of Conduct" on page 119 and in "The MTRR Registers" on page 505.
- **Performance Monitoring Registers.** A description of this enhancement can be found in "Performance Monitoring and Timestamp" on page 437. Please note that the performance monitoring features were added in the Pentium processor, but were only documented with the release of the Pentium Pro processor.
- **Machine Check Architecture MSRs.** A description of this enhancement can be found in "Machine Check Architecture" on page 415.

DebugCTL, LastBranch and LastException MSRs

Introduction

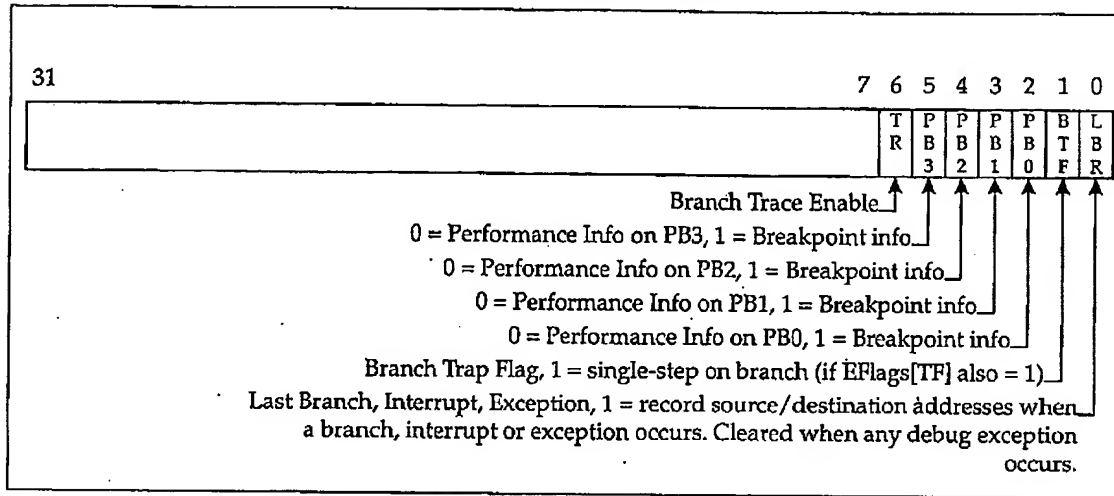
The DebugCTL MSR (see Figure 19-1 on page 375) controls the following processor features related to program debugging:

- Enable/disable recording of source and target addresses related to the last branch, interrupt or exception (see "Last Branch, Interrupt or Exception Recording" on page 375).
- Enable/disable single-step exception on branches, interrupts or exceptions (see "Single-Step Exception on Branch, Exception or Interrupt" on page 376).
- Configure the processor's PMB (Performance Monitoring or Breakpoint) output pins as either breakpoint or performance monitoring event outputs (see "Performance Monitoring and Timestamp" on page 437).
- Enable/disable the processor's ability to generate the branch trace message transaction when a branch is taken (see "Branch Trace Message Transaction Used for Program Debug" on page 340).

Please note that, at the time of this writing, the Intel documentation has discrepancies related to the bit assignment of the DebugCTL MSR. Figure 10-2 on page 10-11 of the Operating System Writer's Manual and the layout of the DebugCTL register as defined in appendix C of the same book do not agree with each other.

Chapter 19: Register Set Enhancements

Figure 19-1: DebugCTL MSR



Last Branch, Interrupt or Exception Recording

When set to one, the LBR (last branch recording; see Figure 19-1 on page 375) bit in the DebugCTL MSR causes the processor to record the source and target addresses of the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. These addresses are recorded in four MSRs provided solely for this purpose:

- **LastBranchToIP** MSR. Records the last address branched to by a branch, exception, or interrupt.
- **LastBranchFromIP** MSR. Records the last address branched from by a branch, exception, or interrupt.
- **LastExceptionToIP** and **LastExceptionFromIP** MSRs. When an exception or interrupt occurs, the processor first copies the LastBranchToIP and LastBranchFromIP registers into these registers before recording the to and from addresses for the exception or interrupt in the LastBranchToIP and LastBranchFromIP registers. They can be thought of as the save registers for the last branch information.

When the processor generates a debug exception, the DebugCTL[LBR] bit is cleared before execution of the debug exception handler, but the contents of the last branch and last exception MSRs is untouched.

Pentium Pro Processor System Architecture

Single-Step Exception on Branch, Exception or Interrupt

See Figure 19-1 on page 375. When the DebugCTL[BTF] (branch trap flag) and the EFLAGS[TF] bits are both set to one, the processor generates a single-step debug exception the next time it takes a branch, exception, or interrupt.

Before entering the debug single-step handler, both the DebugCTL[BTF] and EFLAGS[TF] are cleared, disabling the single-step exception.

New Bits in Pre-Existent Registers

CR4 Enhancements

Refer to Figure 19-2 on page 377. The following feature control bits have been added to CR4:

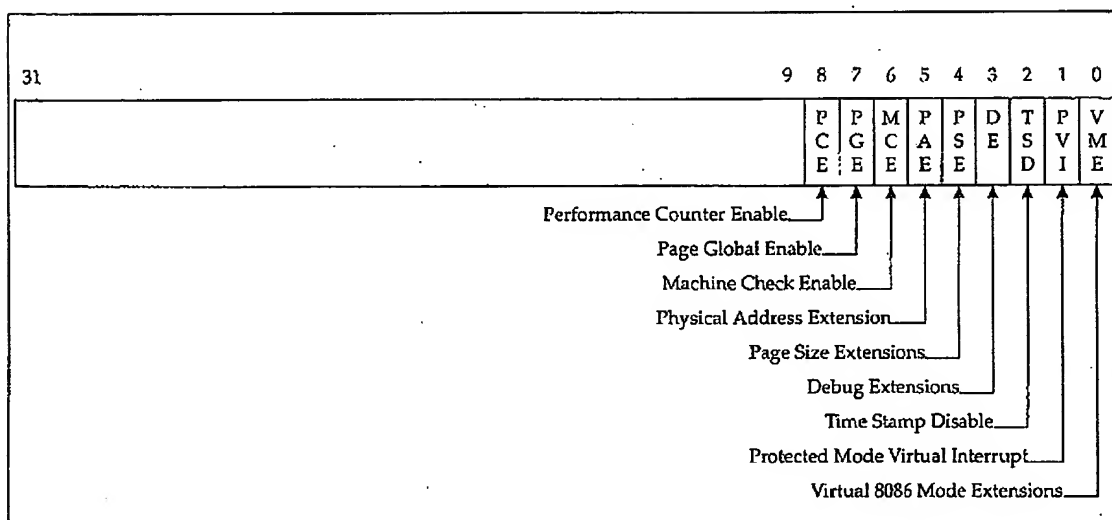
- **VME. Virtual 8086 Mode Extensions.** For a detailed description of this feature, refer to "VM86 Mode Extensions" on page 403. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **PVI. Protected Mode Virtual Interrupts.** For a detailed description of this feature, refer to "Interrupt Enhancements" on page 401. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **TSD. Time Stamp Disable.** For a detailed description of this feature, refer to "Time Stamp Counter Facility" on page 438. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **DE. Debug Extensions.** Please note that this feature was added in the Pentium processor. For a detailed description, refer to the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).
- **PSE. Page Size Extension.** For a detailed description of this feature, refer to "Paging Enhancements" on page 379. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **PAE. Physical Address Extension.** For a detailed description of this feature, refer to "Paging Enhancements" on page 379. *This feature is new in the Pentium Pro processor.*
- **MCE. Machine Check Exception Enable.** For a detailed description of this

Chapter 19: Register Set Enhancements

feature, refer to "Machine Check Architecture" on page 415. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.

- **PGE. Page Global Enable.** For a detailed description of this feature, refer to "Paging Enhancements" on page 379. *This feature is new in the Pentium Pro processor.*
- **PCE. Performance Counter Enable.** For a detailed description of this feature, refer to "Performance Monitoring and Timestamp" on page 437. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.

Figure 19-2: CR4 Feature Bits



CR3 Enhancements

When the CR4 Physical Address Extension (PAE) bit is set to one, CR3 is no longer the Page Directory Base Address register. Rather, it is the Page Directory Pointer Table (PDPT) Base Address register. For a detailed description of the PAE feature, refer to "Paging Enhancements" on page 379.

20 *Paging Enhancements*

The Previous Chapter

The previous chapter described new registers, the bits within them, and the new features that they are associated with.

This Chapter

This chapter describes the following paging-related processor features:

- The **Page Size Extension (PSE)** feature introduced in the Pentium processor. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- The **Physical Address Extension (PAE)** feature introduced in the Pentium Pro processor. This feature permits programs to access code or data within a 64GB rather than a 4GB range. In future processors, the PAE could permit access to up to 2^{64} memory space.
- The **Global Page (PGE)** feature introduced in the Pentium Pro processor.

The Next Chapter

The next chapter describes innovations made in the Pentium Pro processor that are related to interrupts and exceptions. This includes some innovations that first appeared in the Pentium processor, but were not publicly documented until the Pentium Pro was introduced.

Background on Paging

For a detailed background on paging as implemented on x86 processors, refer to the MindShare book entitled *Protected Mode Software Architecture* (published by Addison-Wesley). Please be aware, however, that the discussion of 4MB

Pentium Pro Processor System Architecture

pages in that book has been superseded by the discussion that follows. At the time that the *Protected Mode Software Architecture* book was written, the PSE feature was still undocumented and the description in that book was based on hopefully intelligent speculation. Intel has documented the PSE feature in the Pentium Pro manuals, so we have tuned the description based on this new information.

Page Size Extension (PSE) Feature

The Problem

Consider two examples:

1. There is a rather large memory area that requires identical rules of conduct throughout. Let's say that it is a 1MB video frame buffer area. Without the ability to define large pages, the OS programmer would be forced to set up 256 page table entries, each describing the location and rules of conduct within a 4KB page; in order to cover the entire 1MB memory region. This takes a lot of housework and consumes 1KB of memory (256 entries, each 4 bytes wide) just to describe 256 contiguous 4KB memory regions, each with identical rules of conduct. In addition, since the TLB is typically fairly small, the processor can only cache a subset of these 256 page table entries in its TLB at a given instant in time. As other pages in memory are accessed, the page table entries previously-cached that describe this area are cast out of the TLB to make room for the new page table entries. This results in poor performance the next time one of these pages is accessed. The processor is forced to consult the page directory and page tables in memory in order to refetch the respective page table entry and place it back in the TLB.
2. The OS kernel code (i.e., the core of the OS), is typically kept in memory all of the time and is frequently called by other portions of the OS and by applications programs. The kernel can consume a rather large region of memory. Without large page capability, the OS programmer would have to create and maintain a large number of page table entries, each describing a 4KB area of the OS kernel. As in the previous example, as other pages in memory are accessed, the page table entries previously-cached that describe this area are cast out of the TLB to make room for the new page table entries. This results in poor performance the next time one of these pages is accessed. The processor is forced to consult the page directory and page tables in memory in order to refetch the respective page table entry and place it back in the TLB.

Chapter 20: Paging Enhancements

The Solution—Big Pages

The Pentium and Pentium Pro processors solve this problem as follows:

1. any entry in the Page Directory can be set up as a pointer to a 4MB page in physical memory, rather than a pointer to a page table that describes 4KB pages.
2. the processor implements a separate TLB to cache 4MB entries read from the Page Directory. The first time that an access is made within the 4MB page, the Page Directory entry is cached in the 4MB page TLB. Any subsequent accesses within the same 4MB area then result in a hit on the 4MB TLB, resulting in better performance. In addition, page table entries for 4KB pages are cached in the 4KB page TLB. They do not therefore cause a castout from the 4MB page TLB (or visa versa).

How It Works

This discussion assumes that $CR4[PAE] = 0$ (see Figure 20-1 on page 382) and that paging is enabled ($CR0[PG] = 1$). The PSE feature was first implemented in the Pentium processor and was first publicly documented in the Pentium Pro processor manuals.

Normally, each entry in the Page Directory contains the base address of a Page Table and each entry in that Page Table contains the base physical address of a target page in memory. However, when $CR4[PAE] = 0$ and $CR4[PSE] = 1$, the programmer can set up entries in the Page Directory to point to 4MB physical pages in memory rather than to Page Tables that identify 4KB pages. Refer to Figure 20-2 on page 382. When the PS bit (bit 7 in the Page Directory entry) = 0, the entry contains the base address of a Page Table that describes 4KB pages. However, when the PS bit = 1, the Page Directory entry contains the base address of a 4MB page in memory (see Figure 20-3 on page 383).

Pentium Pro Processor System Architecture

Figure 20-1: CR4

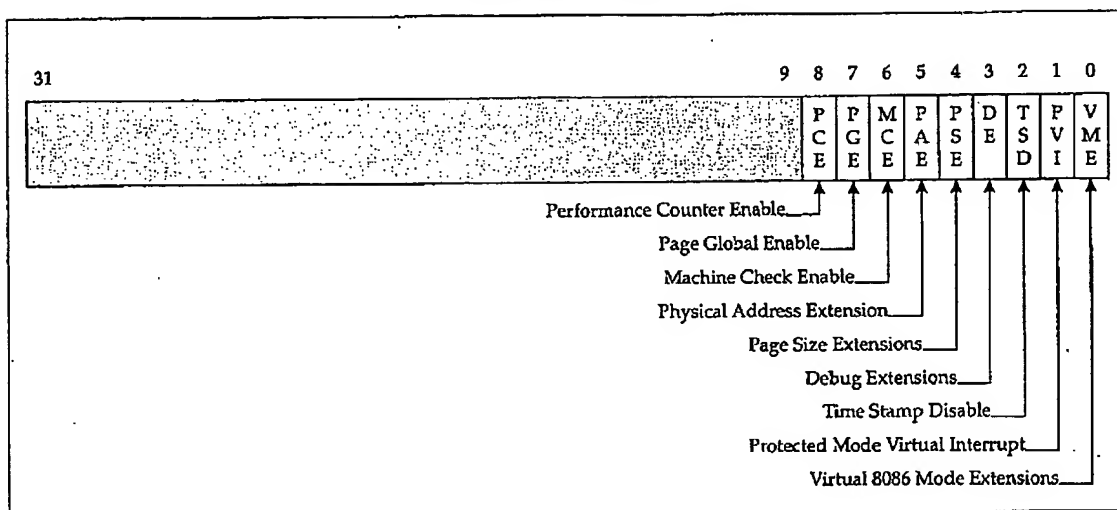
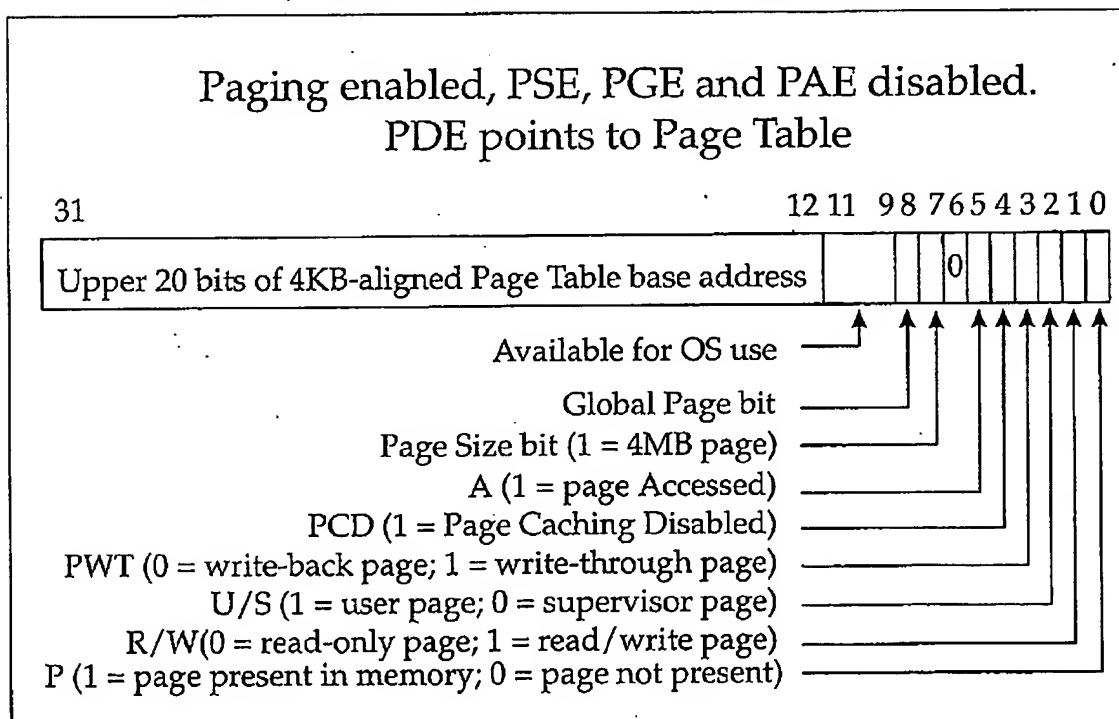
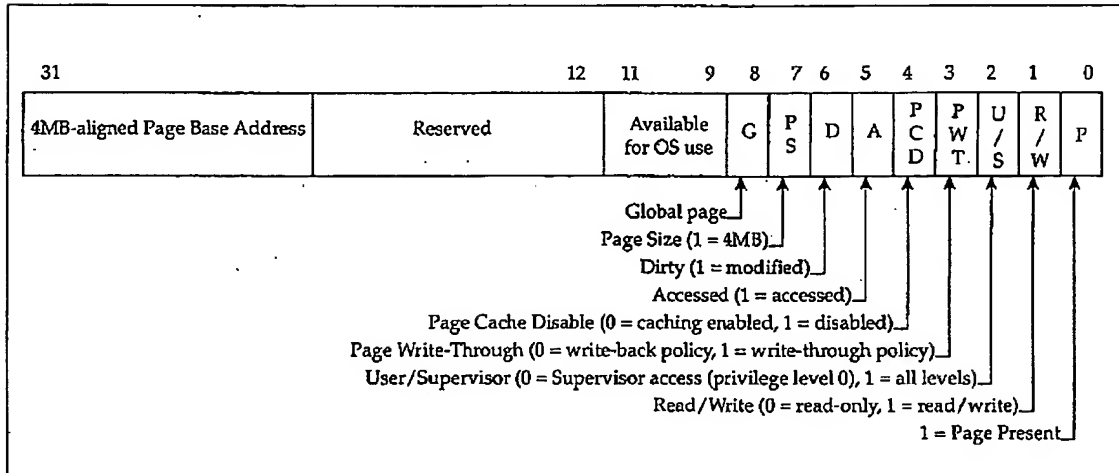


Figure 20-2: Page Directory 32-bit Entry Format (with PAE disabled)



Chapter 20: Paging Enhancements

Figure 20-3: Page Directory Entry for 4MB Page



Physical Address Extension (PAE) Feature

Please note that this entire discussion assumes that $CR4[PAE] = 1$ and $CR4[PSE] = 0$. In other words, the PAE is enabled but the Page Size Extension (PSE) is disabled. Immediately following the discussion of the PAE, "The PAE and the Page Size Extension (PSE)" on page 394 describes how usage of the PSE alters the operation of the PAE.

How Paging Normally Works

Without the PAE feature enabled ($CR4[PAE] = 0$), the processor considers the overall physical memory address space as being 4GB in size, divided into 2^{20} pages of 4KB each. When a memory access is necessary (e.g., instruction fetch, or a memory data read or write caused by the execution of an instruction), the 32-bit memory address generated by adding the segment base address and the 32-bit offset to each other produces a 32-bit address. The program that is currently executing uses 32-bit addressing to address memory, thereby providing the program with the capability of addressing up 4GB of memory space. This is referred to as the linear, or logical, address. When paging is disabled, the linear address is the physical memory address that is read or written.

When the OS initially places a page of code or data related to a particular program in physical memory, it records the start location of the page in a table in

Pentium Pro Processor System Architecture

memory. The program is not informed of the actual location of the page of information in memory. Whenever the currently-executing program attempts to access any location within the page, the processor's paging unit treats the program-generated 32-bit address as a logical address which it must somehow use to index into the Page Tables in memory to discover where the page of information actually resides in physical memory. In other words, it must map or translate the logical page number to the physical page number. Only then can the target physical location within the page be accessed. The paging unit views the 4GB logical address space as subdivided into 1024 page groups, each consisting of 1024 logical pages 4KB in size. The paging unit therefore views the 32-bit linear (i.e., logical) address as being divided into the following fields:

- Bits [31:22] is the **target logical page group** (1-of-1024). This upper 10-bit field of the linear address is used as an index into the Page Directory, selecting an entry that points to a Page Table. The selected Page Table tracks the current physical location of the 1024 pages that make up this page group.
- Bits [21:12] is the **target page within the group** (1-of-1024). This middle 10-bit field of the linear address is used as the index into the Page Table associated with this page group. The entry selected by this bit field keeps track of the current physical location of this page in memory.
- Bits [11:0] is the **start address within the page** for the read or write (1-of-4096).

The OS sets up and maintains the Page Directory and the Page Tables to track the mapping of logical pages-to-physical pages:

- CR3 is loaded with the 4KB-aligned physical base address of the Page Directory.
- Each entry in the Page Directory corresponds to a logical page group and contains the base address of the Page Table associated with that logical page group.
- There is a Page Table for each logical page group.
- Each entry in a Page Table corresponds to a logical page within that logical page group.
- Each Page Table entry contains the 32-bit base physical address of where the logical page of information was placed in memory by the OS at an earlier point in time (when the page of information was created or was loaded into memory from a mass storage device).

Using this address translation mechanism, the processor can map any access generated by the currently-executing program to any location within the 4GB physical memory area.

Chapter 20: Paging Enhancements

What Is the PAE?

The current versions of the Pentium Pro processor have 36 address lines and can address up to 64GB of physical memory. The 386, 486 and Pentium processors only have 32 address lines, permitting a maximum of 4GB of physical memory to be addressed. When the PAE is disabled ($CR4[PAE] = 0$), the 32-bit linear address generated by the currently-executing program is translated into a 32-bit physical memory address. In other words, it's backward-compatible with the earlier processors. If things were left this way, the programmer and the processor are incapable of addressing memory over the 4GB address boundary and the upper four address lines, $A[35:32]\#$, are always deasserted.

Once the PAE is enabled, the processor's page address translation mechanism is redefined, permitting the translation, or mapping, of a 32-bit logical memory address to any location within the processor's 64GB address range (in reality, future x86 processors with address buses up to 64-bits wide are supported by the PAE mechanism).

How Is the PAE Enabled?

The PAE is enabled by setting $CR4[PAE] = 1$. In addition, some changes to the paging data structures must also be made. These changes are defined in the sections that follow.

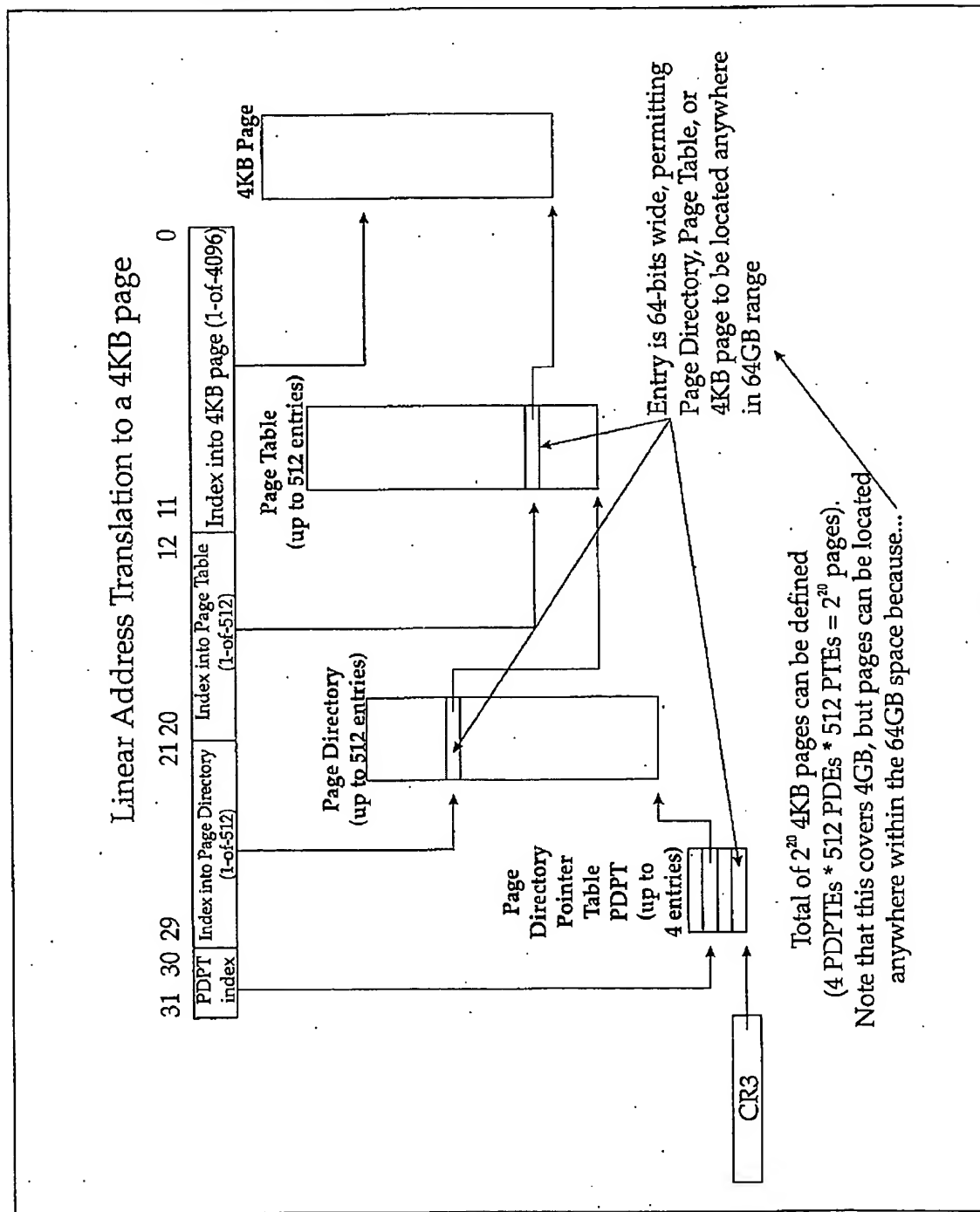
Changes to the Paging-Related Data Structures

Paging without the PAE enabled is described in the section entitled "How Paging Normally Works" on page 383 and in more detail in the MindShare book entitled *Protected Mode Software Architecture* (published by Addison-Wesley). Without the PAE enabled, the OS builds and maintains a two-level directory structure that is automatically interrogated by the processor's paging unit to determine which physical page the target logical page maps to.

Refer to Figure 20-4 on page 386. When the PAE is enabled, the OS must build and maintain a three-level directory structure. A new data structure, the Page Directory Pointer Table, or PDPT, contains the base address of up to four Page Directories. The Page Directory is still required, but, rather than acting as the level one directory, it is the level two directory and the Page Tables are the level three directories.

Pentium Pro Processor System Architecture

Figure 20-4: Paging Directory Structure when PAE Enabled



Chapter 20: Paging Enhancements

Programmer Still Restricted to 32-bit Addresses and 2^{20} Pages

The programmer can still only generate 32-bit addresses for memory data reads, memory data writes, or instruction fetches. In addition, the new 3-level directory structure can still only keep track of the location of up to 2^{20} pages of information (512 pages per Page Table * 512 Page Tables per Page Directory * 4 Page Directories).

Pages Can be Distributed Throughout Lower 64GB

Refer to Figure 20-4 on page 386. However, this directory structure permits the 32-bit address to be mapped to any page within the 64GB address space that the Pentium Pro is capable of addressing:

- **Without PAE enabled**—each Page Directory and Page Table entry is 32-bits wide with a 20-bit field to identify the 4KB-aligned base address of a Page Table or a page within the lower 4GB. The lower 12 bits of the linear address is added to the lower end of this 20-bit field, creating the exact address of one of the locations within the target physical 4KB page.
- **With PAE enabled**—each Page Directory and Page Table entry is 64-bits wide with a 24-bit field to identify the 4KB-aligned base address of a Page Table or a page within the lower 64GB. The lower 12 bits of the linear address is added to the lower end of this 24-bit field, creating the exact address of one of the locations within the target physical 4KB page.

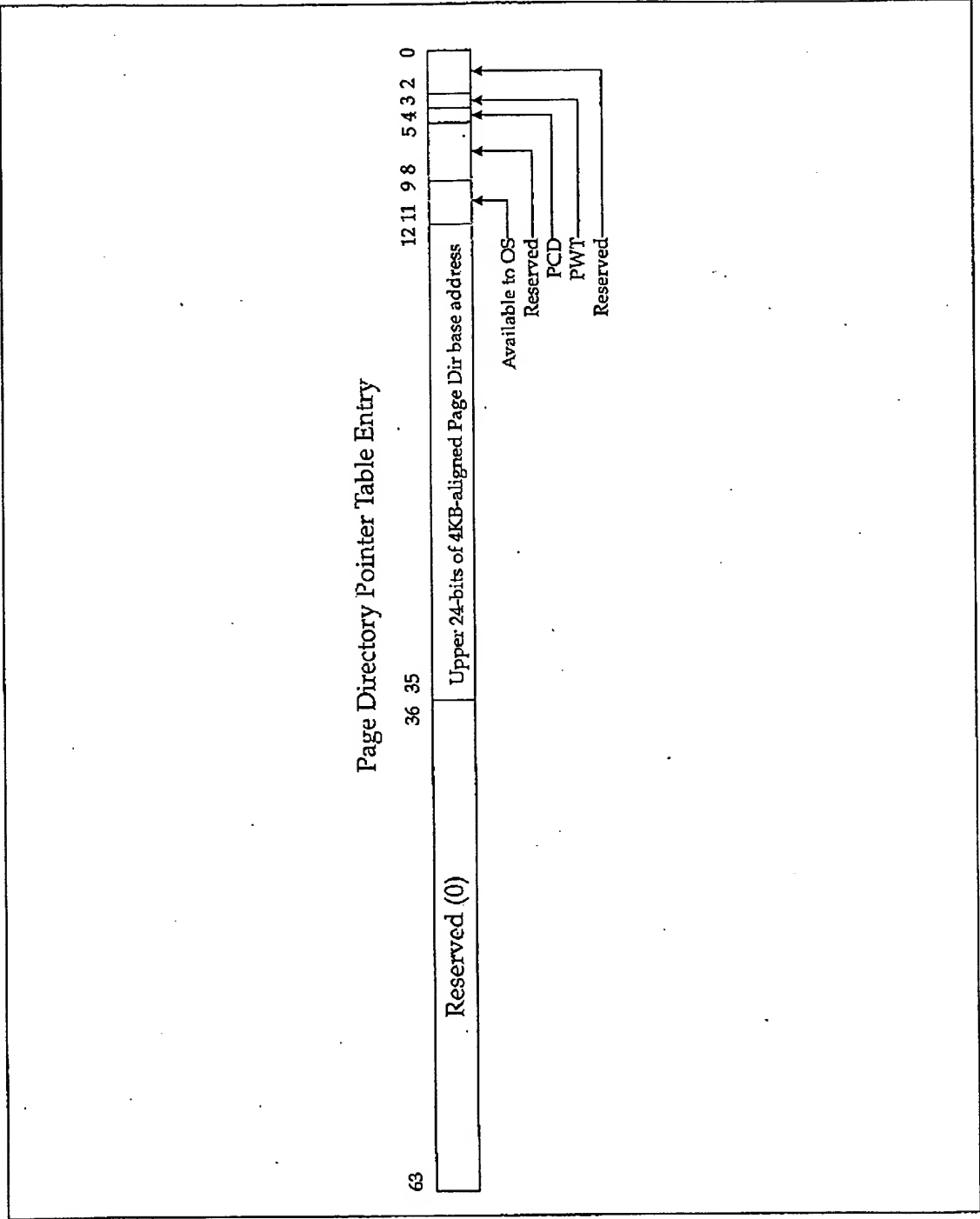
CR3 Contains Base Address of PDPT

Figure 20-5 on page 388 illustrates the contents of CR3 when the PAE is enabled. It contains the base address of the Page Directory Pointer Table (PDPT). Notice that it is only a 32-bit register. This means that the PDPT must reside within the lower 4GB of memory. The programmer loads bits [31:5] of the PDPT base address, meaning that it must be 32-byte-aligned (the entire PDPT fits in one cache line).

When the CR4[PAE] and CR0[PG] bits are set to one, or, if they are already both set and CR3 is loaded with a new value, the processor automatically reads all four PDPT entries into an invisible register set inside the processor. These val-

Chapter 20: Paging Enhancements

Figure 20-6: Format of a PDPT Entry



Pentium Pro Processor System Architecture

Format of Page Directory Entry

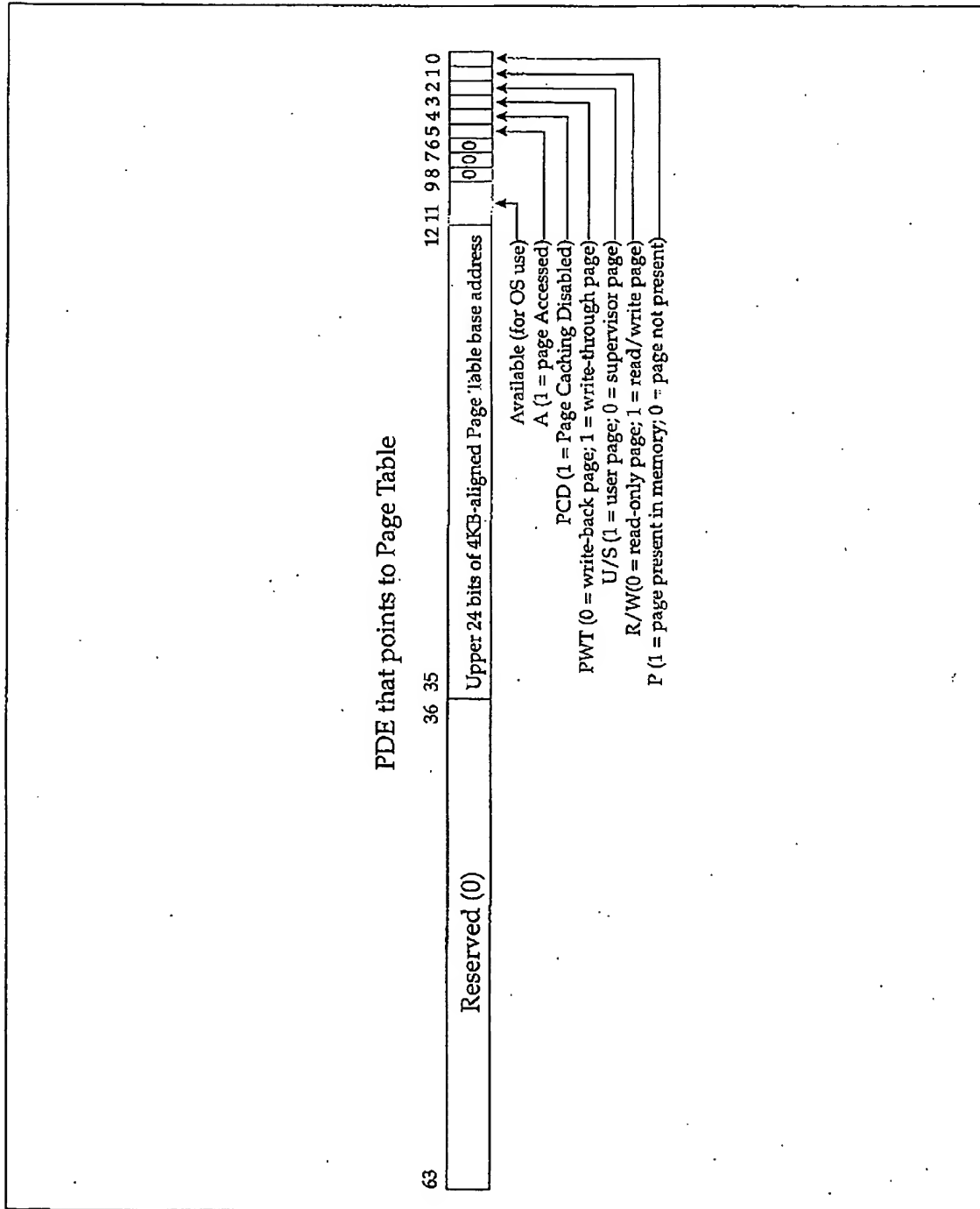
Figure 20-7 on page 391 illustrates the format of a Page Directory entry. The reserved bits must all be zero. The programmer loads the entry with the 4KB-aligned (the upper 24-bits of the) address of the Page Table. This permits the Page Table to be located on any one of 2^{24} 4KB-aligned address boundaries within the lower 64GB.

The state of the PCD (Page Cache Disable) and PWT (Page Write-Through) bits tells the processor whether the Page Table is cacheable and, if it is, whether to use a write-through or a write-back policy in handling writes to the Page Table. The remaining bits have the following meaning (in the following discussion, note that the supervisor privilege levels are 0, 1, and 2):

- **P** is the **Page Present** bit. The OS programmer sets $P = 1$ if the Page Table is currently present in memory. In addition, the programmer indicates its location in memory using the base address field and sets the attribute bits (PCD, PWT, R/W, U/S, and A) to the appropriate states. If the Page Table is not currently present in memory, $P = 0$ and bits [63:1] of the entry can be used by the OS for anything. Any memory access that selects the Page Table will then result in a page fault exception and the 32-bit linear address that caused the fault is recorded in CR2.
- **R/W** is the **Read/Write** bit. When $R/W = 0$, all of the pages pointed to by the Page Table are read-only. When $R/W = 1$, read/write access is permitted within those pages. Used in conjunction with the U/S bit (see next bullet in list). In addition, if $CR0[WP] = 1$, programs executing at supervisor privilege level must also obey this bit. If $CR0[WP] = 0$, programs executing at the supervisor privilege level don't have to obey this bit.
- **U/S** is the **User/Supervisor** bit. When $U/S = 0$, only programs executing at the supervisor privilege level can access the pages pointed to by the Page Table. When $U/S = 1$, the pages can be accessed by programs executing at any privilege level. Note that access rights are further qualified by the R/W bit.
- **A** is the **Accessed** bit. Set by the processor the first time that the Page Table is accessed (for a read or a write).

Chapter 20: Paging Enhancements

Figure 20-7: Format of Page Directory Entry that Points to Page Table



Pentium Pro Processor System Architecture

Format of Page Table Entry

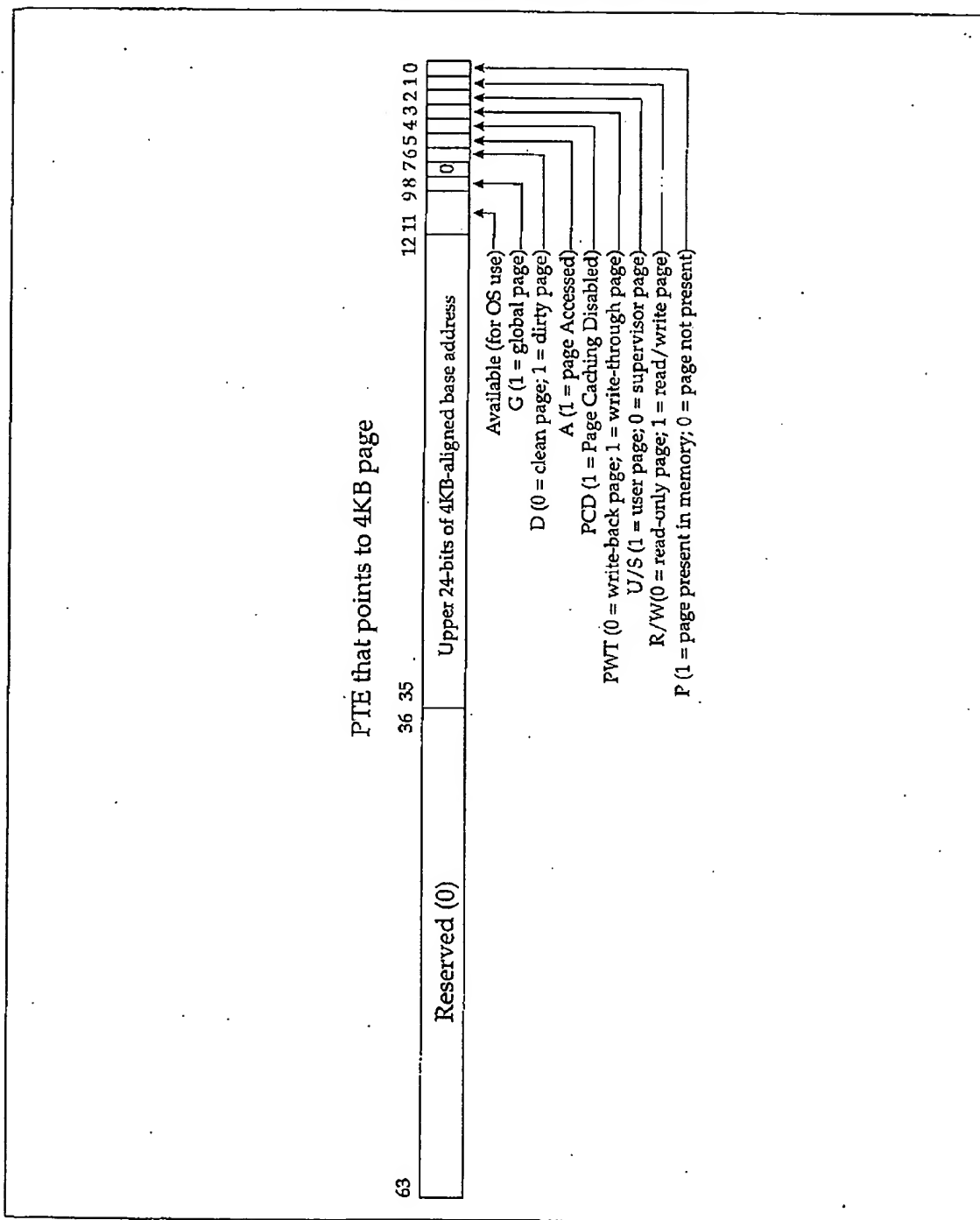
Figure 20-8 on page 393 illustrates the format of a Page Table entry. The reserved bits must all be zero. The programmer loads the entry with the 4KB-aligned (the upper 24-bits of the) address of the page in memory. This permits the page to be located on any one of 2^{24} 4KB-aligned address boundaries within the lower 64GB.

The state of the PCD (Page Cache Disable) and PWT (Page Write-Through) bits tells the processor whether the page is cacheable and, if it is, whether to use a write-through or a write-back policy in handling writes to the page. The remaining bits have the following meaning (in the following discussion, note that the supervisor privilege levels are 0, 1, and 2):

- **P** is the **Page Present** bit. The OS programmer sets $P = 1$ if the page is currently present in memory. In addition, the programmer indicates its location in memory using the base address field and sets the attribute bits (PCD, PWT, R/W, U/S, D, G and A) to the appropriate states. If the page is not currently present in memory, $P = 0$ and bits [63:1] of the entry can be used by the OS for anything. Any memory access that selects the page will then result in a page fault exception and the 32-bit linear address that caused the fault is recorded in CR2.
- **R/W** is the **Read/Write** bit. When $R/W = 0$, the page is read-only. When $R/W = 1$, read/write access is permitted within the page. Used in conjunction with the U/S bit (see next bullet in list). In addition, if $CR0[WP] = 1$, programs executing at supervisor privilege level must also obey this bit. If $CR0[WP] = 0$, programs executing at the supervisor privilege level don't have to obey this bit.
- **U/S** is the **User/Supervisor** bit. When $U/S = 0$, only programs executing at the supervisor privilege level can access the page pointed to by this Page Table entry. When $U/S = 1$, the page can be accessed by programs executing at any privilege level. Note that access rights are further qualified by the R/W bit.
- **A** is the **Accessed** bit. Set by the processor the first time that the page is accessed (for a read or a write).
- **D** is the **Dirty** bit. Set by the processor the first time that page is written to.
- **G** is the **Global** bit. When $G = 1$, the page is identified as a global page. The processor should retain this page table entry in its TLB when a task switch occurs (i.e., when a new value is loaded into CR3). For a more detailed discussion, refer to "Global Page Feature" on page 398.

Chapter 20: Paging Enhancements

Figure 20-8: Format of Page Table Entry



Pentium Pro Processor System Architecture

The PAE and the Page Size Extension (PSE)

Now assume that both the PAE and the PSE bits in CR4 are set to one. With one exception, everything that has been described about the PAE feature remains true. The difference lies in the format and usage of Page Directory entries. Normally, each Page Directory entry contains the base address of a Page Table and each entry of the Page Table contains the base address of a 4KB page in memory.

When the PSE and PAE features are both activated, however, the PS (Page Size) bit (bit 7) in a directory entry can now be set to one without causing a general protection exception. When PS = 1, the Page Directory entry doesn't point to a Page Table. Rather, it contains the 2MB-aligned physical base address of a 2MB page in memory and the attribute bits that defined the rules of conduct within that page.

Figure 20-9 on page 396 illustrates the translation of a 32-bit linear address to a 2MB page somewhere in the lower 64GB. Figure 20-10 on page 397 illustrates the format of Page Directory entry that identifies a 2MB page in memory and the rules of conduct that must be followed when performing accesses within the page.

The reserved bits must all be zero. The programmer loads the entry with the 2MB-aligned (the upper 15-bits of the) address of the page in memory. This permits the page to be located on any one of 2^{15} 2MB-aligned address boundaries within the lower 64GB.

The state of the PCD (Page Cache Disable) and PWT (Page Write-Through) bits tells the processor whether the page is cacheable and, if it is, whether to use a write-through or a write-back policy in handling writes to the page. The remaining bits have the following meaning (in the following discussion, note that the supervisor privilege levels are 0, 1, and 2):

- P is the **Page Present** bit. The OS programmer sets P = 1 if the page is currently present in memory. In addition, the programmer indicates its location in memory using the base address field and sets the attribute bits (PCD, PWT, R/W, U/S, D, G and A) to the appropriate states. If the page is not currently present in memory, P = 0 and bits [63:1] of the entry can be used by the OS for anything. Any memory access that selects the page will then result in a page fault exception and the 32-bit linear address that caused the fault is recorded in CR2.
- R/W is the **Read/Write** bit. When R/W = 0, the page is read-only. When

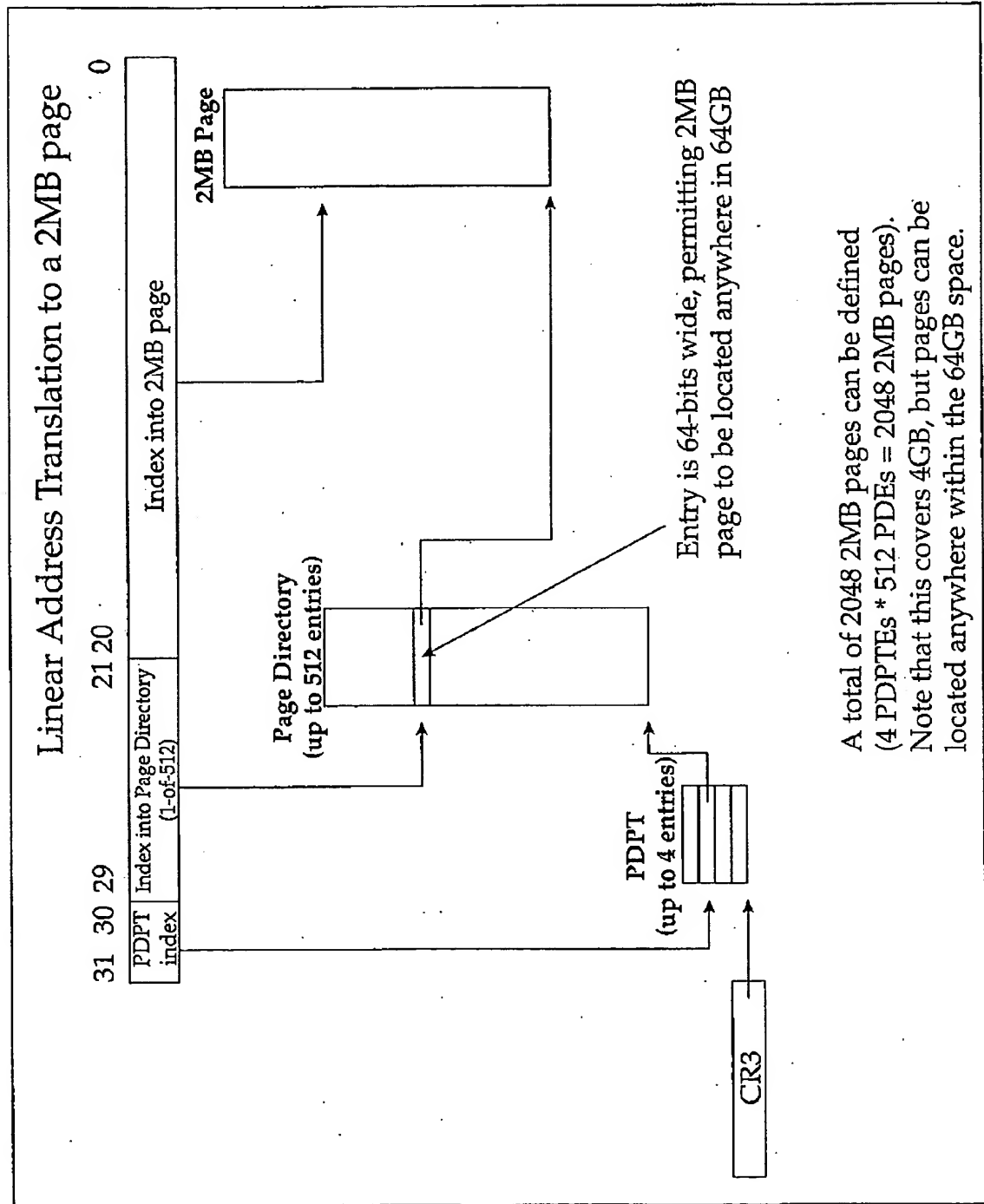
Chapter 20: Paging Enhancements

R/W = 1, read/write access is permitted within the page. Used in conjunction with the U/S bit (see next bullet in list). In addition, if CR0[WP] = 1, programs executing at supervisor privilege level must also obey this bit. If CR0[WP] = 0, programs executing at the supervisor privilege level don't have to obey this bit.

- U/S is the **User/Supervisor** bit. When U/S = 0, only programs executing at the supervisor privilege level can access the page pointed to by this Page Table entry. When U/S = 1, the page can be accessed by programs executing at any privilege level. Note that access rights are further qualified by the R/W bit.
- A is the **Accessed** bit. Set by the processor the first time that the page is accessed (for a read or a write).
- D is the **Dirty** bit. Set by the processor the first time that page is written to.
- G is the **Global** bit. When G = 1, the page is identified as a global page. The processor should retain this page table entry in its TLB when a task switch occurs (i.e., a new value is loaded into CR3). For a more detailed discussion, refer to "Global Page Feature" on page 398.

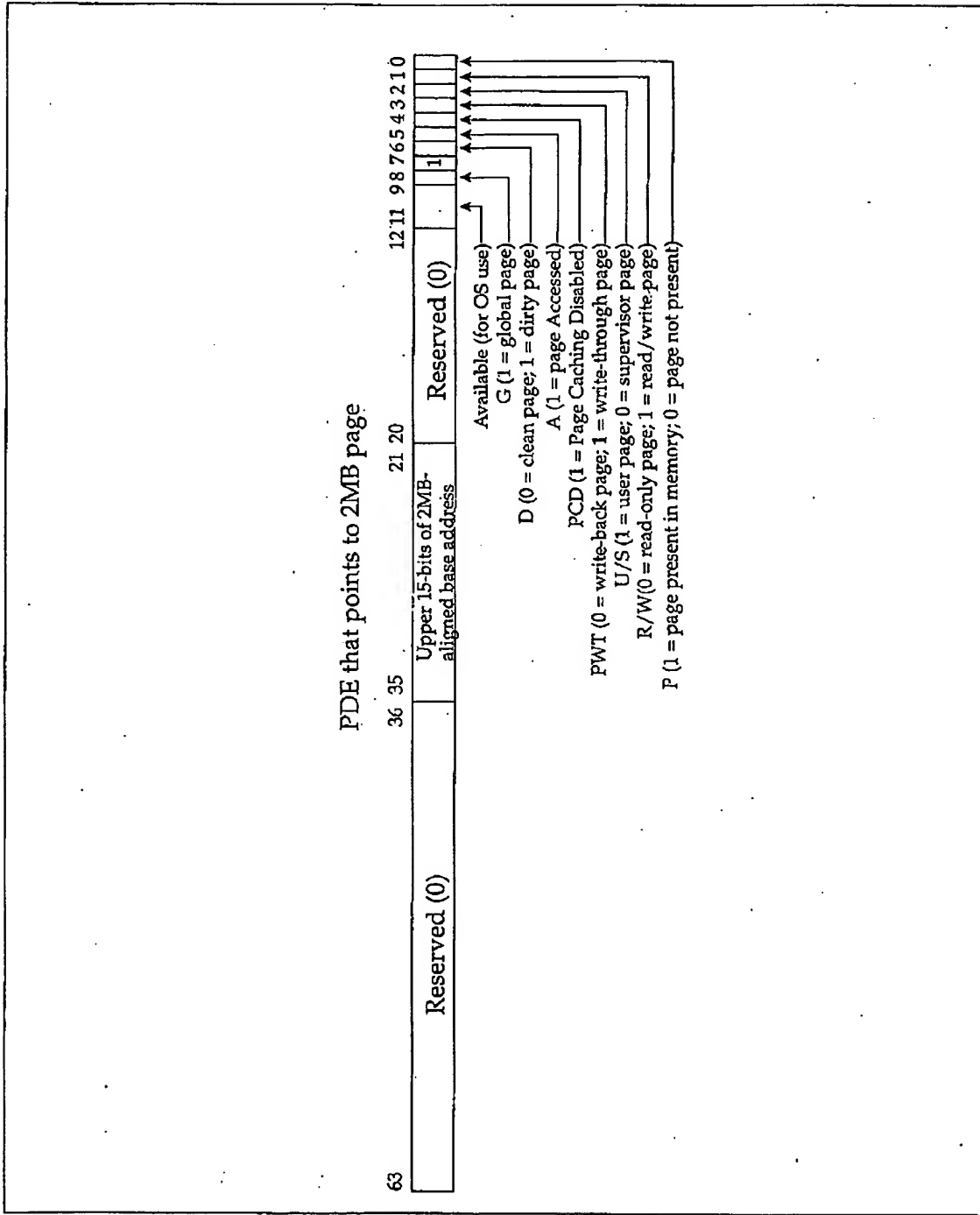
Pentium Pro Processor System Architecture

Figure 20-9: Translation of Linear Address to a 2MB Page



Chapter 20: Paging Enhancements

Figure 20-10: Format of a Page Directory Entry for 2MB Page



Pentium Pro Processor System Architecture

Global Page Feature

The Problem

During normal operation, the processor copies page table entries related to pages being accessed into a special cache known as the TLB (Translation Lookaside Buffer). This results in faster performance. Once the processor has placed a page's Page Table entry in the TLB, the processor has the logical-to-physical page information on-chip. It doesn't have to read the Page Directory and then the Page Table to obtain this information every time an access is made to the page.

Whenever a task switch occurs, the processor loads a new value into CR3, selecting the set of Page Directories and Page Tables that define the pages accessible by the new task. When this occurs, the processor automatically flushes all entries currently-cached in the TLB (because they were cached from the previous program's Page Tables). This causes a performance dip as the new program starts to execute because the processor will have to consult the multi-level directories in memory to obtain copies of the Page Table entries related to this program.

Quite commonly, a number of pages are shared among multiple programs. It would therefore be more efficient if the processor retained the Page Table entries related to these "global", or shared, pages in the TLB rather than discarding them along with the rest when a task switch occurs.

The Solution

The Pentium Pro processor implements a new feature referred to as the Global Page feature. It is activated by setting CR4[PGE] = 1. Once activated, any Page Table entries (or Page Directory entries for 2MB pages) that have the G bit = 1 (see Figure 20-8 on page 393 and Figure 20-10 on page 397) are considered global. When a task switch is performed, all of the TLB entries are flushed with the exception of those marked global. The specification states that these entries will remain in the TLB indefinitely. This is a very "fuzzy" term. It more than likely means that a global entry will remain in the TLB until it is cast out to make room for a new Page Table entry. As time goes on and accesses are made to new pages, the processor reads the Page Table entries for the newly-accessed pages and places their contents into the TLB. If the TLB entry selected by the TLB's

Chapter 20: Paging Enhancements

LRU (least-recently used) algorithm is already occupied by a previously-cached Page Table entry, the older entry is cast out of the TLB to make room for the more recently accessed entry.

The specification states that the only way to deterministically (i.e., definitely) remove the global entries from the TLB is to set CR4[PGE] = 0 and then invalidate the TLB. The TLB is invalidated by clearing one of the following control bits:

- CR0[PE] to leave protected mode.
- CR0[PG] to disable paging.
- CR4[PSE] to disable the Page Size Extension feature.
- CR4[PGE] to disable the Global Page feature.
- CR4[PAE] to disable the Physical Address Extension.

21 *Interrupt Enhancements*

The Previous Chapter

The previous chapter described the following paging-related processor features:

- The **Page Size Extension (PSE)** feature introduced in the Pentium processor. Please note that this feature was added in the Pentium processor, but was only documented with the release of the Pentium Pro processor.
- **Physical Address Extension (PAE)** feature introduced in the Pentium Pro processor. This feature permits programs to access code or data within a 64GB rather than a 4GB range.
- The **Global Page (PGE)** feature introduced in the Pentium Pro processor.

This Chapter

This chapter describes innovations made in the Pentium Pro processor that are related to interrupts and exceptions. This includes some innovations that first appeared in the Pentium processor, but were not publicly documented until the Pentium Pro was introduced.

The Next Chapter

The next chapter describes the error logging and reporting machine check architecture first introduced in the Pentium processor and greatly expanded in the Pentium Pro processor. The machine check architecture defines an exception (the machine check exception) used to report hardware-related problems and a register set used to log both recoverable and unrecoverable errors.

Pentium Pro Processor System Architecture

New Exceptions

The Pentium Pro does not implement any new software exception conditions.

Added APIC Functionality

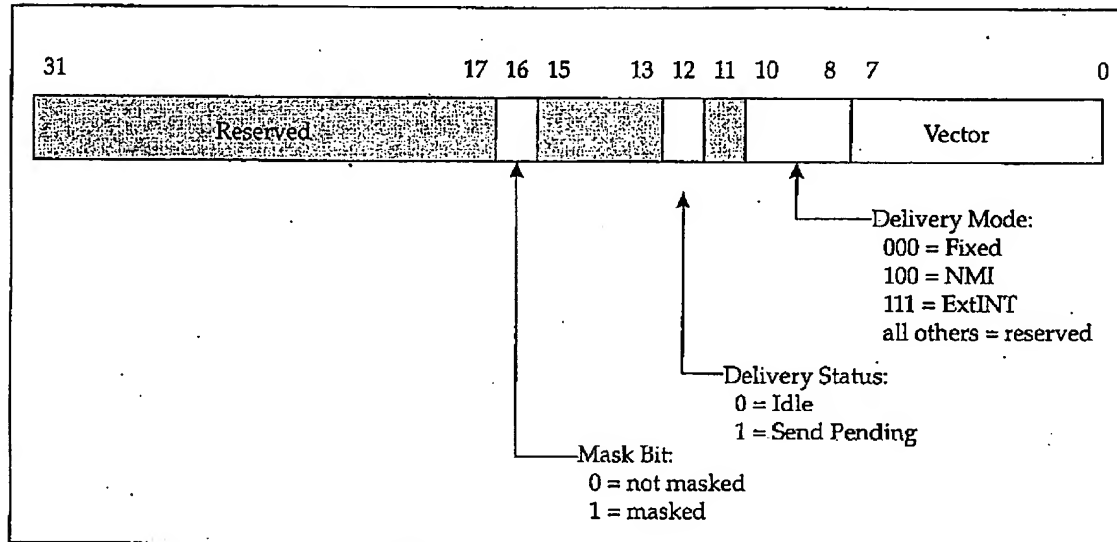
Either or both of the performance counters (see "Performance Monitoring and Timestamp" on page 437) may be programmed to generate an interrupt when the respective event counter overflows. The local APIC design has been enhanced by adding a fifth entry to its local vector table, or LVT (for information on local APIC operation and its LVT, refer to the chapter on the APIC in MindShare's book entitled *Pentium Processor System Architecture*, published by Addison-Wesley).

Figure 21-1 on page 403 illustrates the format of the performance counter entry in the APIC's LVT. The bit fields within the entry have the following usage:

- the **vector field** identifies the entry in the Interrupt Descriptor Table (IDT) that contains the pointer to the performance counter overflow interrupt handler routine that will be executed each time that one of the performance counters overflows.
- the **delivery mode field** specifies how the interrupt will be generated: either by jumping through the IDT entry specified in the vector field; or via the NMI entry in the IDT (entry 2) and the vector field will be ignored. Although the specification also shows ExtINT as a valid choice, it is the author's opinion this isn't so. If the processor were to generate an Interrupt Acknowledge bus transaction to obtain the performance counter vector from the 8259A interrupt controller, it would confuse the 8259A (because it did not generate the interrupt request).
- the **delivery status bit** indicates whether the interrupt has either not been generated or the interrupt has occurred and the handler has already been called (in both cases, the state would be Idle); or the interrupt has occurred, but the handler is still in the process of being called.
- the **mask bit** permits the performance counter overflow event to be ignored (mask = 1) or to be recognized (mask = 0).

Chapter 21: Interrupt Enhancements

Figure 21-1: Format of the Performance Counter LVT Entry



VM86 Mode Extensions

VM86 Mode Background

For a detailed description of virtual 8086 (VM86) mode, refer to the MindShare book entitled *Protected Mode Software Architecture* (published by Addison-Wesley). However, the section entitled *VM86 Mode Extensions* in that book was written based on speculation (since Intel had not yet released the documentation of the VM86 Mode Extensions). This section is based on the documentation provided by Intel in the *Pentium Pro Operating System Writer's Manual* and takes the place of that discussion.

Interrupt-Related Problems and VM86 Tasks

The sections that follow describe the problems associated with interrupt handling when the processor is executing a VM86 task (typically a DOS task) under a multitasking OS in VM86 mode (EFLAGS[VM] = 1). The software overhead imposed by the OS (and the attendant performance hit) in the following scenarios are described:

Pentium Pro Processor System Architecture

- the handling of an attempted execution of a CLI or an STI instruction when the VM86 extensions feature bit, CR4[VME], isn't enabled.
- when the VM86 task attempts to execute a software interrupt instruction (INT *n*) with the VM86 extensions feature bit, CR4[VME], disabled.

Software Overhead Associated with CLI/STI Execution

Attempted Execution of CLI by VM86 Task. Assume that the VM86 task has attempted to disable recognition of external hardware interrupts (because it doesn't want to be bothered by interrupts during execution of a critical piece of code). The processor did not successfully execute the instruction (it resulted in a GP exception because the VM86 task has insufficient privilege), so interrupt recognition is still enabled. The GP exception handler passes control to the VMM (Virtual Machine Monitor) program which is responsible for "baby-sitting" VM86 tasks. There are three possible cases:

1. the VMM checks the state of the IF bit in the EFlags image on the privilege level 0 stack and determines that interrupt recognition had already been disabled (by the VMM or OS) at some earlier point in time. In this case, the VMM adjusts the return pointer on the privilege level 0 stack to point to the instruction following the CLI that caused the exception, and then executes an IRET to resume execution of the interrupted VM86 task at the instruction that follows the CLI.
2. the VMM may know that this is a safe time to disable interrupt recognition (because there are no high-priority interrupts expected). In this case, the VMM could choose to execute a CLI instruction, adjust the return pointer on the privilege level 0 stack to point to the instruction following the CLI that caused the exception, and then execute an IRET to resume execution of the interrupted VM86 task at the instruction that follows the CLI.
3. the VMM may know that this not a safe time to disable recognition of hardware interrupts. The text that follows provides a detailed description of this case.

The multitasking OS cannot permit the VM86 task (which doesn't know of the existence of the multitasking OS or other, currently-suspended tasks) to summarily disable interrupt recognition. At an earlier point in time, another task may have stimulated an IO device (e.g., a disk interface) to perform an operation and generate an interrupt when it has been completed. The device may complete the requested operation and generate the interrupt while the VM86 task is executing. Furthermore, the device may be quite sensitive to being serviced on a timely basis. The VM86 task is unaware of any of this.

Chapter 21: Interrupt Enhancements

Based on the attempted execution of the CLI instruction, the VMM will note (in a memory location somewhere) that the currently-executing task prefers not to be interrupted. In other words, the VMM maintains a virtual copy of EFLAGS[IF] bit in software. It alters the return address on the privilege level 0 stack to point to the instruction that follows the CLI and then executes the IRET to resume execution of the interrupted VM86 task at the instruction that follows the CLI.

If a hardware interrupt should subsequently occur, the VM86 task is interrupted and the hardware interrupt's protected mode handler then passes control to the VMM. If the VMM knows that the interrupting device requires fast servicing, it immediately executes the device's interrupt handler to service the device. In other words, it ignores the preference of the VM86 program that it not be interrupted. In this case, the VM86 task was interrupted even though it preferred not to be. The VMM designer should make every attempt to accomplish the check just described as expeditiously as possible and return control to the interrupted task. Otherwise, the interrupted VM86 task may not function correctly (because of the lengthy delay imposed by the VMM's software overhead necessary to determine whether to service the hardware interrupt right away or to defer servicing it until the task's timeslice has expired).

On the other hand, the VMM may determine that the interrupting device can stand some delay in being serviced and note that the state of the virtual copy of the EFLAGS[IF] bit indicates that the VM86 task prefers not to be interrupted. In this case, the VMM would set a bit in a VMM-specific data structure (let's call it the deferred interrupt data structure) indicating that the specified interrupt handler should be executed when the VM86 task completes its timeslice. It then executes the IRET instruction to resume execution of the interrupted VM86 task. Later, when the VM86 task's timeslice has expired and a task switch occurs back to the OS, the OS checks the deferred interrupt data structure (mentioned earlier) to determine if the servicing of any hardware interrupt(s) was deferred. If it was, the OS calls the respective interrupt handler(s) to service the hardware device(s).

Attempted Execution of STI Instruction. If the VM86 task attempts to reenable interrupt recognition, one of three cases is true:

1. the VMM checks the state of the IF bit in the EFLAGS image on the privilege level 0 stack and determines that interrupt recognition is already enabled. In this case, the VMM adjusts the return pointer on the privilege level 0 stack to point to the instruction following the STI that caused the exception, and then executes an IRET to resume execution of the interrupted VM86 task at the instruction that follows the STI.

Pentium Pro Processor System Architecture

2. the VMM may know that this is a safe time to enable interrupt recognition. In this case, the VMM could choose to execute a **STI** instruction, adjust the return pointer on the privilege level 0 stack to point to the instruction following the **STI** that caused the exception, and then execute an **IRET** to resume execution of the interrupted VM86 task at the instruction that follows the **STI**.
3. the VMM knows that this is not a safe time to reenale recognition of hardware interrupts. In this case, the VMM adjusts the return pointer on the privilege level 0 stack to point to the instruction following the **STI** that caused the exception, and then executes an **IRET** to resume execution of the interrupted VM86 task at the instruction that follows the **STI**.

Servicing of Software Interrupts by DOS or OS

Many VM86 tasks utilize the **INT *nn*** instruction (where *nn* selects an entry in the interrupt table in memory) to call the real mode OS or BIOS services. An attempt to execute an **INT *nn*** instruction when the **EFLAGS[IOPL]** field < 3 results in a GP exception (due to insufficient privilege). The processor executes the protected mode GP exception handler. The handler checks the VM bit in the **EFLAGS** image on the privilege level 0 stack to determine if the interrupted task is a VM86 task. If it is (**VM** = 1), the GP handler passes control to the VMM. The VMM must then determine what to do in response. There are two basic cases:

1. The VMM determines that it is not legal for the VM86 task to call the target vector. In this case, the VMM would be forced to terminate the VM86 task.
2. The VMM determines that the VM86 task is attempting to call a DOS or BIOS service.

In the second case, the VMM must choose one of the following options:

1. Pass the call to the respective real mode handler.
2. Pass the call to the protected mode OS to handle. When the OS has completed the request, the VMM then returns control to the interrupted VM86 task at the instruction immediately following the **INT *nn*** instruction.

Solution—VM86 Mode Extensions

Introduction

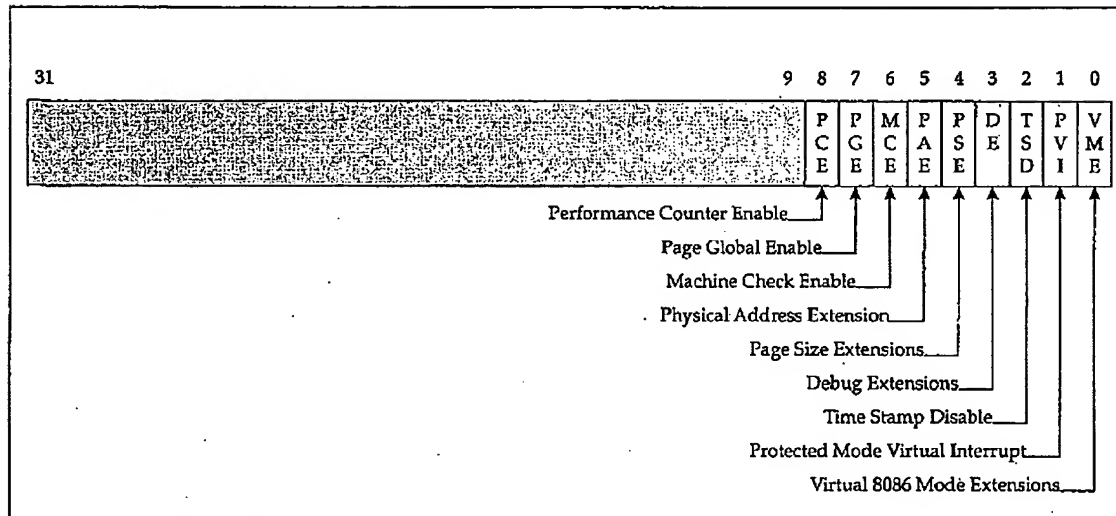
The VM86 mode extensions are enabled by setting **CR4[VME]** = 1 (see Figure 21-2 on page 407). Once this bit is enabled, the following bits are usable:

Chapter 21: Interrupt Enhancements

- **EFLAGS[VIPL]**. Virtual Interrupt Pending bit. See Figure 21-3 on page 408.
- **EFLAGS[VIF]**. Virtual Interrupt Flag bit. See Figure 21-3 on page 408.
- **CR4[PVI]**. Protected Mode Virtual Interrupts. See Figure 21-2 on page 407.

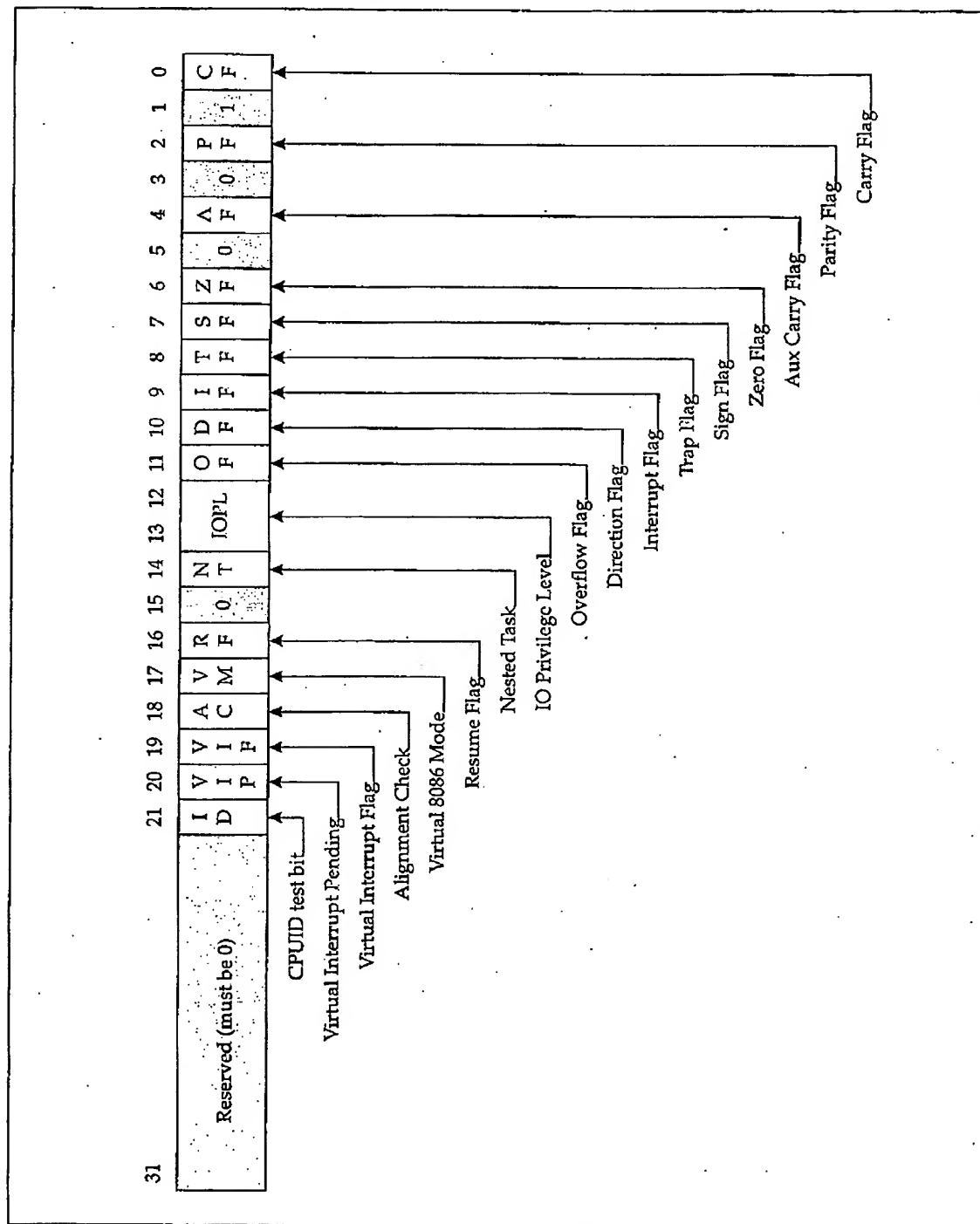
The sections that follow describe how these bits are used to solve the problems associated with interrupt handling in VM86 tasks.

Figure 21-2: CR4



Pentium Pro Processor System Architecture

Figure 21-3: EFLAGS Register



Chapter 21: Interrupt Enhancements

CLI/STI Solution

All VM86 tasks execute at privilege level 3. The effects of executing a CLI or STI instruction while in VM86 mode depend on a number of factors. There are three cases:

1. If $EFLAGS[IOPL] = 3$, the instruction is successful in changing the state of the $EFLAGS[IF]$ bit (because privilege level 3 programs are permitted to execute CLI and STI). Execution of CLI causes recognition of external interrupts to be disabled. Execution of the VM86 task continues.
2. If $EFLAGS[IOPL] < 3$ and $EFLAGS[VME] = 0$, an attempt to execute either CLI or STI causes a GP exception (due to insufficient privilege).
3. If $EFLAGS[IOPL] < 3$ and $EFLAGS[VME] = 1$, execution of either CLI or STI changes the state of $EFLAGS[VIF]$ rather than $EFLAGS[IF]$, and execution of the VM86 task continues. Execution of a CLI instruction clears $EFLAGS[VIF]$ to 0, while execution of the STI instruction sets it to one. The VMM is not called, thus avoiding the software overhead associated with CLI or STI execution. The sections that follow describe what occurs when a hardware interrupt occurs.

$EFLAGS[VIF] = 1$, $EFLAGS[IF] = 1$, Interrupt Occurs. Recognition of external interrupts is enabled ($EFLAGS[IF] = 1$) and the VM86 program has indicated that it doesn't mind being interrupted (as indicated by $EFLAGS[VIF] = 1$).

Any hardware interrupt will cause the processor to call the respective protected mode handler.

$EFLAGS[VIF] = 0$, $EFLAGS[IF] = 1$, Interrupt Occurs. Recognition of external interrupts is enabled ($EFLAGS[IF] = 1$), but the VM86 program has indicated that it wishes not to be interrupted (as indicated by $EFLAGS[VIF] = 0$). The following actions are taken:

1. Processor suspends the VM86 task and calls the protected mode handler associated with the interrupt vector supplied by the interrupt controller.
2. In the protected mode handler, the programmer checks the state of the VM bit in the $EFLAGS$ image saved on the stack. If set, a VM86 task was interrupted, so the VMM program is called and provided with the vector number that corresponds to the hardware interrupt event.
3. In the VMM, check the state of $EFLAGS[VIF]$. In this example scenario, it is cleared, indicating that the VM86 task prefers not be interrupted.
4. The VMM sets $VIP = 1$ in the $EFLAGS$ image on the stack, indicating that the interrupt request for service has been deferred until such time as the

Pentium Pro Processor System Architecture

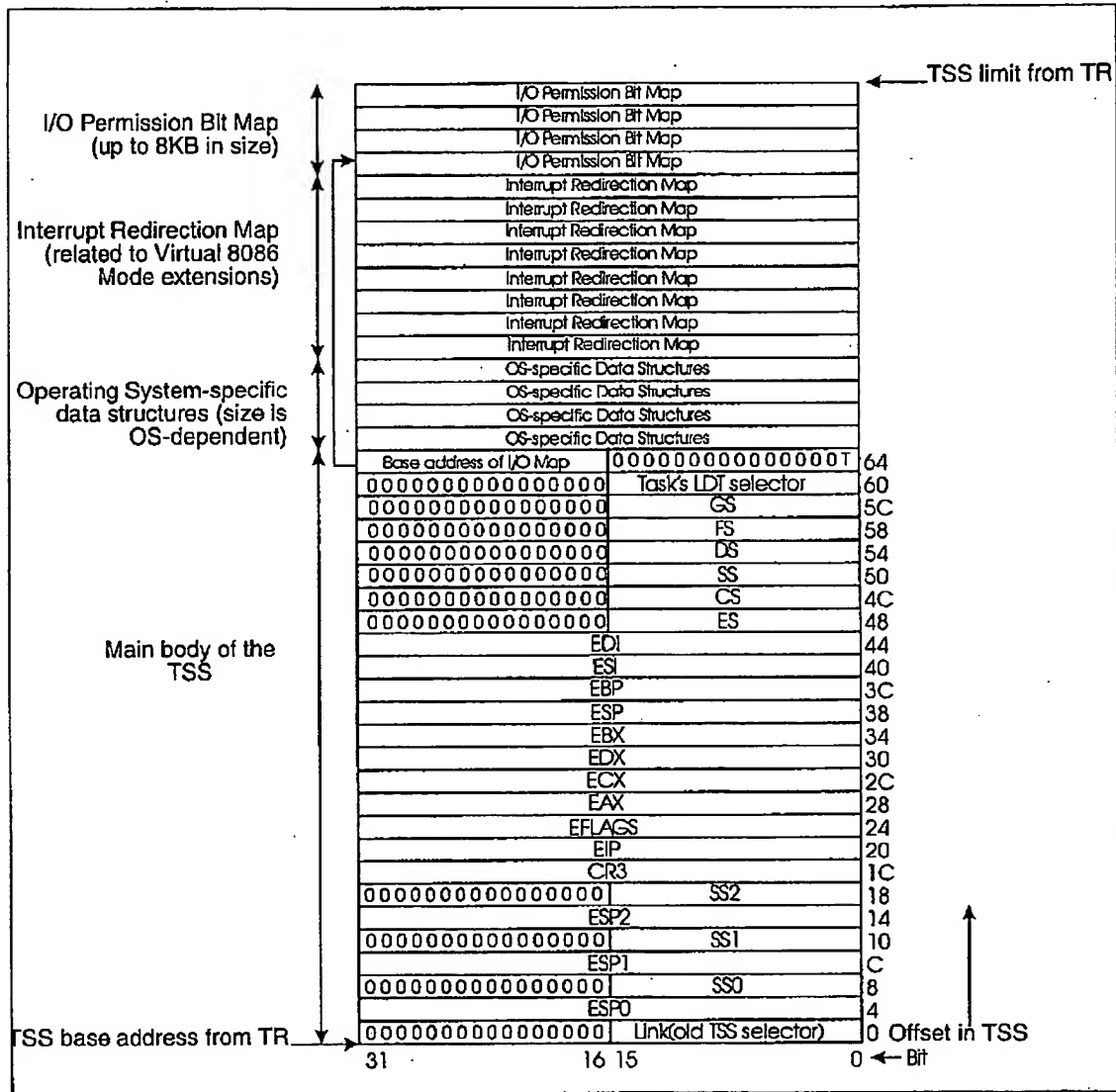
- VM86 task reenables interrupt recognition (by executing an STI instruction), or the VM86 task's time slice is up.
5. The VMM makes note of which handler must be executed at a later time (VMM typically maintains a bitmap corresponding to all interrupt vectors 32d through 255d).
 6. The VMM returns to the protected mode handler that called it.
 7. The handler returns to the interrupted VM86 task. When the processor pops the EFLAGS image from the stack, EFLAGS[IF] = 1 (recognition of hardware interrupts is enabled), EFLAGS[VIF] is still 0 (indicating that the VM86 task prefers not to be interrupted) and EFLAGS[VIP] = 1 (indicating that servicing of an interrupt has been deferred), reenabling interrupt recognition.
 8. When the VM86 task subsequently executes an STI instruction to reenable interrupt recognition (in other words, it no longer minds being interrupted), the processor takes the following actions:
 - Checks the state of EFLAGS[VIP]. If VIP = 0, indicating that no interrupts were deferred, the processor sets EFLAGS[VIF] = 1 (because STI is being executed) and continues execution of the VM86 task. If VIP = 1, this indicates that handling of one or more interrupts was deferred. The processor generates a GP exception.
 - When the GP exception handler detects VIP = 1 in the EFLAGS image on the stack, it calls the VMM (because servicing of one or more interrupts was deferred).
 - The VMM calls the handlers associated with the interrupt (or interrupts) that had been deferred.
 - When the interrupt handler(s) have all been called, the VMM sets VIF = 1 (because an STI instruction is being executed) and clears VIP = 0 in the EFLAGS image on the stack (because all deferred interrupts have been serviced).
 - The VMM then returns to the VM86 task.

Software Interrupt Redirection Solution

Hardware interrupts are always handled by the protected mode handlers, as are NMI and software exceptions. The interrupt redirection bitmap in the TSS (see Figure 21-4 on page 411) is not consulted. The processor only consults the interrupt redirection bitmap when executing software interrupt instructions. Table 21-1 on page 412 assumes that a software interrupt instruction is being executed in a VM86 task. There are six different scenarios that define how it is handled.

Chapter 21: Interrupt Enhancements

Figure 21-4: Task State Segment (TSS)



Pentium Pro Processor System Architecture

Table 21-1: VM86 Interrupt/Exception Handling

Scenario	VME Bit	EFLAGS IOPL bit field	Redir Map Bit	Description of Processor/Handler Actions
1	0	3	x	<p>When a software interrupt occurs, the processor switches from VM86 mode to protected mode and executes protected mode handler.</p> <ul style="list-style-type: none"> Switches to privilege level 0 stack. Pushes data segment registers onto level 0 stack (so they can be restored to their original values when resuming execution of interrupted program). Clears data segment registers. Pushes SS, ESP, EFLAGS, CS, and EIP of interrupted task onto level 0 stack. If set, clears EFLAGS[TF], disabling single-step. Clears EFLAGS[VM]. Processor exits VM86 mode. If vector selects interrupt gate, clears EFLAGS[IF], disabling recognition of external interrupts. Sets CS:EIP from interrupt gate descriptor selected by vector and begins execution of protected mode handler.
2	0	<3	x	<p>Attempted execution of any software interrupt instruction causes a GP exception due to insufficient privilege. GP handler can then handle the event.</p>
3	1	<3	1	<p>When a software interrupt occurs, the processor consults the interrupt redirection bitmap in the task's TSS to determine whether to vector to the DOS handler or to the GP exception handler. In this case, the redirection bit = 1, indicating the GP exception handler will be called.</p>
4	1	3	1	<p>Same as scenario 1 (i.e., processor switches from VM86 mode to protected mode and executes appropriate protected mode handler).</p>

Chapter 21: Interrupt Enhancements

Table 21-1: VM86 Interrupt/Exception Handling (Continued)

Scenario	VME Bit	EFLAGS IOPL bit field	Redir Map Bit	Description of Processor/Handler Actions
5	1	3	0	<p>Software interrupt is redirected to 8086 handler:</p> <ul style="list-style-type: none"> • Pushes CS:IP onto current task's stack. • Clear EFLAGS[NT] and EFLAGS[IOPL]. • Push Flag onto current task's stack. • If set, clears EFLAGS[TF], disabling single-step. • If set, clears EFLAGS[IF], disabling recognition of external interrupts. • Loads CS:IP from real mode interrupt table entry selected by vector. Real mode interrupt table starts at linear address 0 in the current task. • Begins execution of real mode interrupt handler. <p>Execution of a CLI or STI affects the EFLAGS[IF] (because privilege level 3 programs are permitted to execute CLI and STI).</p>
6	1	<3	0	<p>Same as scenario 5, with the following additional effect on hardware interrupt handling. When CR4[VME] = 1, the CPL of the DOS task is 3, and EFLAGS[IOPL] < 3, attempted execution of CLI and STI instructions changes the state of EFLAGS[VIF] (rather than EFLAGS[IF]). See "CLI/STI Solution" on page 409.</p>

Virtual Interrupt Handling in Protected Mode

The CLI/STI handling mechanism (just described for VM86 mode) can also be used when executing a protected mode task (in other words, not a VM86 task). To activate this capability, CR4[PVI] = 1 and CR4[VME] = 0. This will allow privilege level 3 protected mode programs to execute CLI/STI without causing a GP exception (when IOPL < 3), and prevents the protected mode application program from successfully playing with the EFLAGS[IF] bit.

22 *Machine Check Architecture*

The Previous Chapter

The previous chapter described innovations made in the Pentium and Pentium Pro processors that are related to interrupts and exceptions. This includes some innovations that first appeared in the Pentium processor, but were not publicly documented until the Pentium Pro was introduced.

This Chapter

This chapter describes the error logging and reporting Machine Check Architecture first introduced in the Pentium processor and greatly expanded in the Pentium Pro processor. The Machine Check Architecture defines an exception (the Machine Check exception) used to report hardware-related problems and a register set used to log both recoverable and unrecoverable errors.

The Next Chapter

The next chapter describes the performance monitoring and time stamp counter facilities incorporated in the Pentium Pro processor. It should be noted that these facilities are also implemented in the Pentium processor, but Intel did not document them until the release of the Pentium Pro processor.

Purpose of Machine Check Architecture

The Machine Check Architecture exception and register set logs and reports hardware errors associated with:

- bus errors
- ECC errors
- parity errors

Pentium Pro Processor System Architecture

- cache errors
- TLB errors

As described later in this chapter, the Machine Check Architecture register set is implemented as a set of error logging register banks. It is Intel's intention that these registers be used in the following manner:

- The OS incorporates a utility (i.e., a daemon) that is executed on a periodic basis tasked with scanning the Machine Check Architecture error logging register banks on a regular basis. Any errors that have been logged by the processor(s) are then saved in some form of non-volatile memory. This error log can then be viewed by system maintenance personnel using a special utility program.
- Whenever a processor generates a Machine Check exception, the Machine Check exception handler scans the register banks to determine the cause of the exception. It then determines whether or not the condition can be corrected and, if it can, logs the error in non-volatile memory (see previous bullet), corrects the condition, and resumes execution of the interrupted program. Please note that as of this writing, none of the error conditions that result in a machine check exception are correctable.

Machine Check Architecture in the Pentium Processor

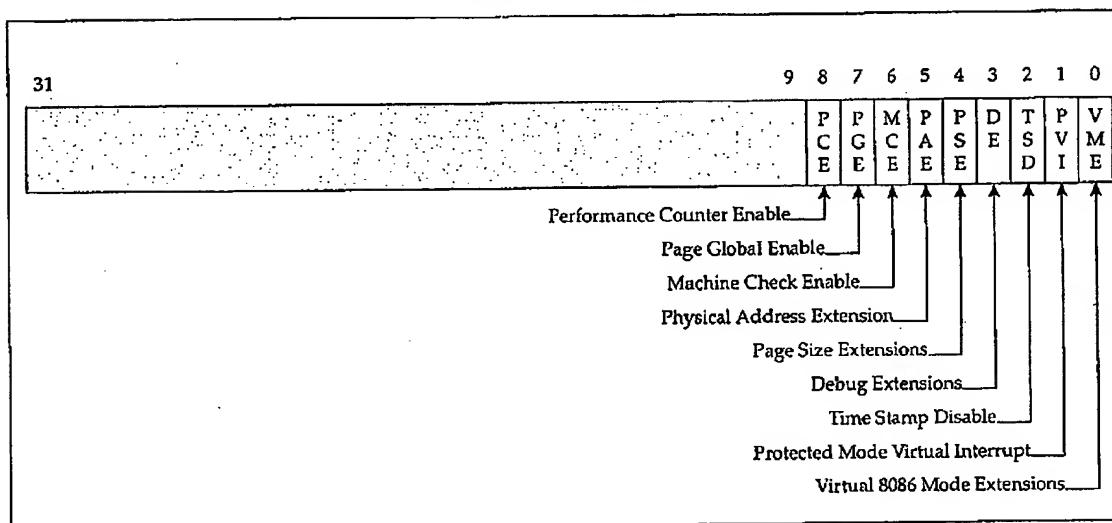
The Pentium processor included a very limited implementation of the Machine Check Architecture. This consisted of:

- The 64-bit Machine Check address register, or MCAR, implemented as a MSR. Latches the address related to a bus cycle where a parity error is detected on a read and PEN# is asserted by the external logic. Also latches the address related to a bus cycle that is aborted by the assertion of the BUSCHK# signal to the processor.
- The 64-bit Machine Check type register, or MCTR, implemented as a MSR. Latches the bus cycle type when a parity error is detected on a read and PEN# is asserted by the external logic. Also latches the bus cycle type that is aborted by the assertion of the BUSCHK# signal to the processor.
- The Machine Check exception (exception 18d). The processor calls the Machine Check handler if a read parity error or a BUSCHK# is detected during a bus cycle and the Machine Check exception has been enabled by CR4[MCE] = 1.
- The Machine Check exception enable bit, CR4[MCE] (see Figure 22-1 on page 417). When set, enables the processor to generate a Machine Check exception (exception 18d) when a read parity error or a BUSCHK# is detected during a bus cycle.

Chapter 22: Machine Check Architecture

- **PEN#** (parity enable) input signal. When the processor reads one or more bytes over the data bus and detects a parity error, it asserts its PERR# output. If the chipset decides that a read from the target address should have yielded correct parity information, it asserts PEN# back to the processor. The processor latches the address and bus cycle type into the MCAR and MCTR, respectively, and, if enabled to do so (CR4[MCE] = 1), also generates a Machine Check exception.
- **BUSCHK#** (bus check) input signal. If the chipset generates BUSCHK# in response to a bus cycle generated by the processor, the processor latches the address and bus cycle type into the MCAR and MCTR, respectively, and, if enabled to do so (CR4[MCE] = 1), also generates a Machine Check exception.

Figure 22-1: CR4

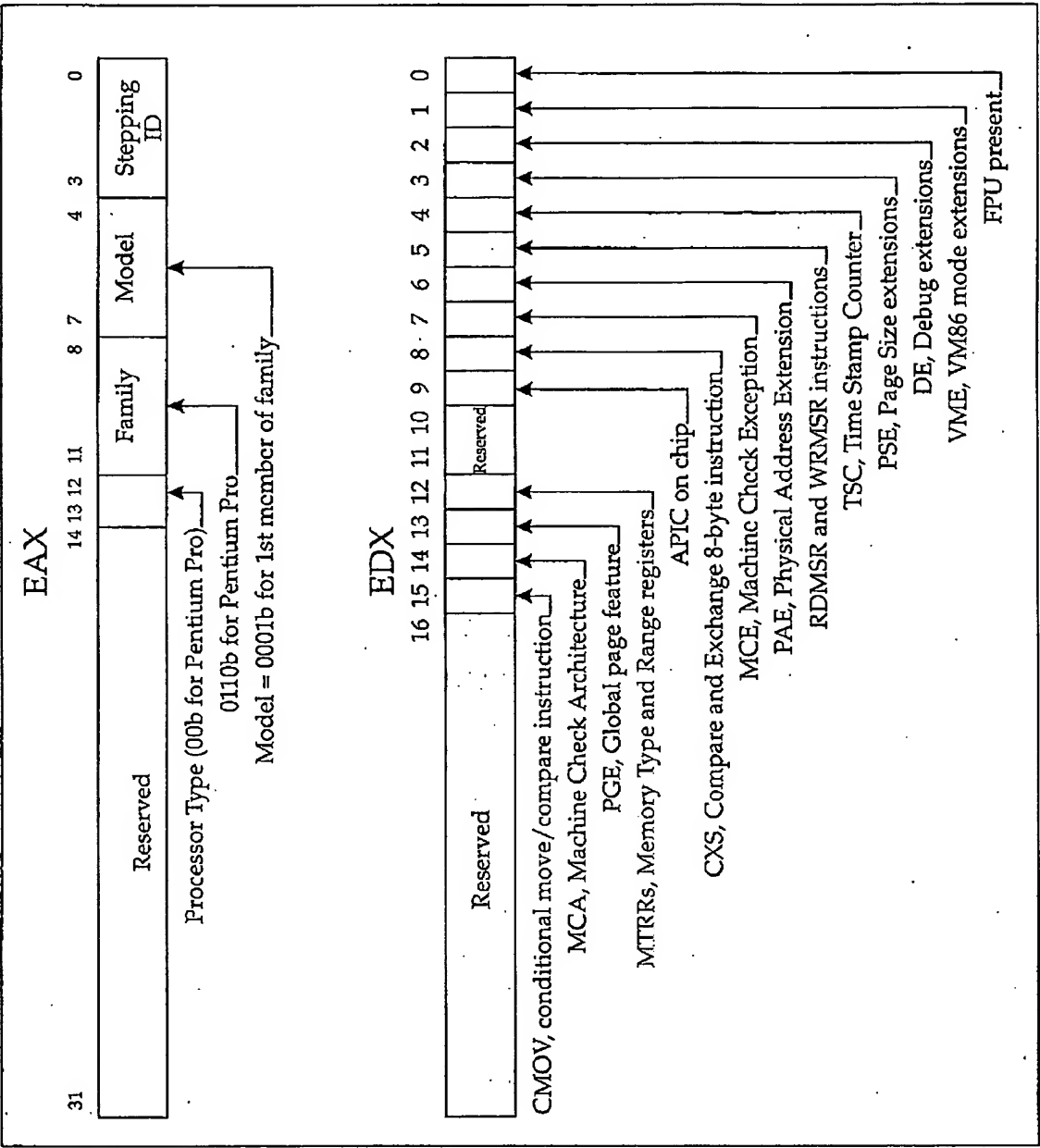


Testing for Machine Check Support

The programmer may determine if a processor supports the Machine Check exception and the Machine Check Architecture register set by executing the CUID instruction with a request for feature support information. The information illustrated in Figure 22-2 on page 418 is returned. EDX[7] = 1 indicates that the Machine Check exception is supported, and EDX[14] = 1 indicates that the Machine Check Architecture register set is supported.

Pentium Pro Processor System Architecture

Figure 22-2: Feature Support Information Returned by CPUID



Chapter 22: Machine Check Architecture

Machine Check Exception

The Machine Check exception is enabled by setting CR4[MCE] = 1. It uses entry 18d in the interrupt descriptor table (IDT). If it is disabled and a hardware failure occurs that would ordinarily result in the generation of a Machine Check exception, the processor generates the special transaction to broadcast the shutdown message and then enters the shutdown state (in other words, it freezes).

The Machine Check exception is an abort class exception. This means that the program that was interrupted by the exception may not be reliably resumed by executing an IRET instruction at the end of the Machine Check exception handler.

Machine Check Architecture Register Set

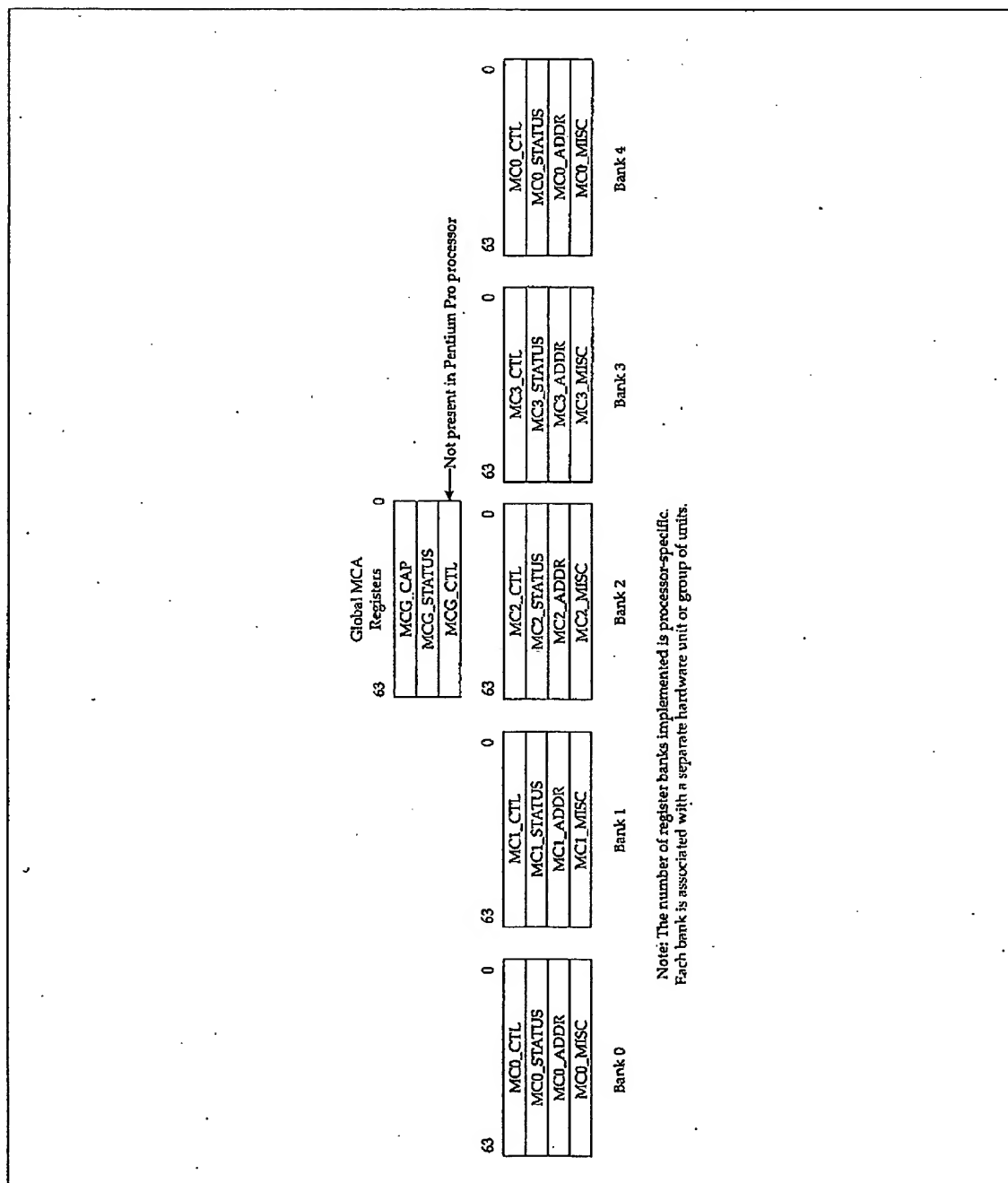
The Machine Check Architecture register set is implemented as a set of MSRs, organized as follows:

- One set of global registers is related to all conditions that may cause a Machine Check exception.
- A separate bank of error logging registers associated with each of the processor's units (or with a group of units). As an example, one register bank may be associated with the external bus unit, while another might be associated with the processor's cache).

The number of register banks implemented and the processor unit(s) that each bank is associated with is processor implementation-specific. As an example, the current implementation of the Pentium Pro processor implements five registers banks (pictured in Figure 22-3 on page 420). Intel does not identify the processor unit(s) that each of these banks is associated with. This bothered the author at first, until it was realized that the error code types reported in each bank reveals this information (because information embedded within the error code field identifies the guilty party).

Pentium Pro Processor System Architecture

Figure 22-3: Pentium Pro Machine Check Architecture Registers



Chapter 22: Machine Check Architecture

Composition of Global Register Set

The global register set consists of the following registers:

- **MCG_CAP** register. Machine Check Global Count and Present register.
- **MCG_STATUS** register. Machine Check Global Status register.
- **MCG_CTL** register. Machine Check Global Control register.

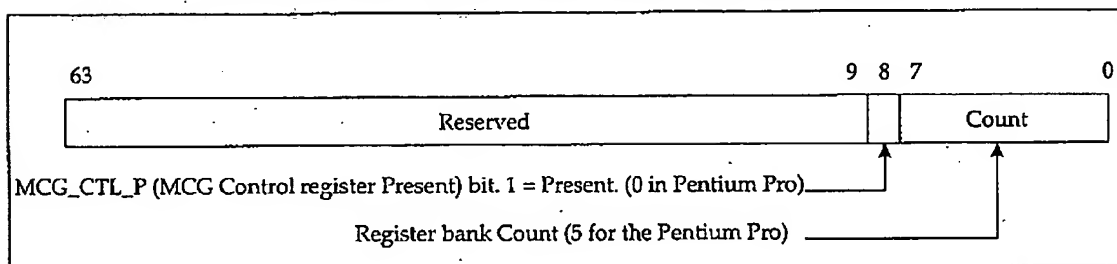
The sections that follow describe each of these registers.

MCG_CAP Register

The Machine Check Global Count and Present register (pictured in Figure 22-4 on page 421) is a read-only register that identifies:

- the number of register banks implemented (minus one). As an example, the value 04h is returned for the current versions of the Pentium Pro processor, indicating that it implements banks 0 through 4.
- whether or not the MCG_CTL register is implemented (it is not implemented in the current versions of the Pentium Pro processor).

Figure 22-4: MCG_CAP (Global Count and Present) Register



MCG_STATUS Register

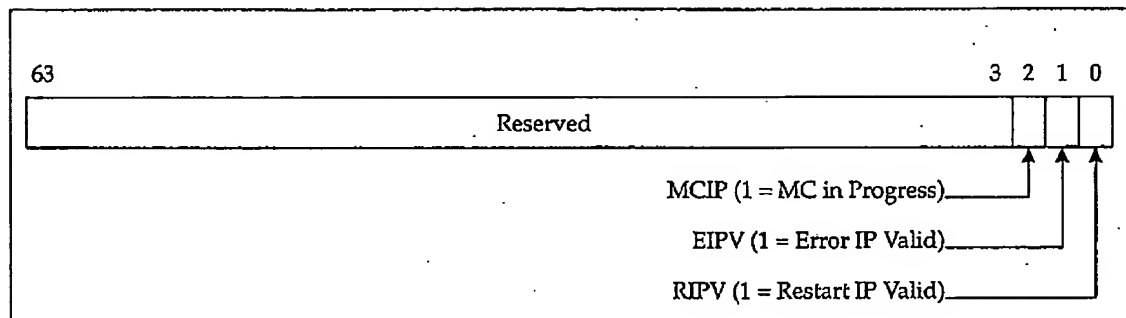
The Machine Check Global Status register (pictured in Figure 22-5 on page 422) indicates the basic state of the processor after a Machine Check exception has occurred. The conditions reported are:

- The RIPV (Restart Instruction Pointer Valid) bit indicates whether or not the program interrupted by the exception can be reliably resumed at the instruction whose pointer was pushed onto the stack when the exception occurred. Most conditions that result in a Machine Check exception preclude the resumption of the interrupted program.

Pentium Pro Processor System Architecture

- The EIPV (Error Instruction Pointer Valid) bit indicates whether the instruction whose pointer was pushed onto the stack when the exception occurred is directly associated with the exception. As an example, a transaction generated on the external bus as a result of the execution of an IO read instruction (IN) might result in a Machine Check (if there's a serious problem detected during the transaction). In this case, the address of the IN instruction might be pushed onto the stack.
- The MCIP (Machine Check In Progress) bit is set to one by the processor when a Machine Check exception occurs. If another Machine Check occurs while this bit is still set (in other words, the Machine Check handler has not executed to the point where the programmer has cleared the MCIP bit), the processor generates a special transaction to broadcast the shutdown message and then enters the shutdown state.

Figure 22-5: MCG_STATUS Register



MCG_CTL Register

The Machine Check Global Control register may or may not be implemented in a processor (it's *not implemented in the current versions of the Pentium Pro processor*). Its presence (or absence) is indicated via the MCG_CTL_P bit in the MCG_STATUS register (see Figure 22-5 on page 422). The current state of the 64-bits in the MCG_CTL register globally enables or disables the processor's ability to generate a machine check exception for all types of hardware failure conditions. The current specification only supports writing all ones or all zeros to this register—in other words, mass enable or disable of all errors. The control register associated with each register bank controls the generation of the machine check exception for errors related to the processor unit(s) covered by the respective bank. Future versions of x86 processors may permit selective enabling of machine check exception generation for various types of errors and disabling of others.

Chapter 22: Machine Check Architecture

Composition of Each Register Bank

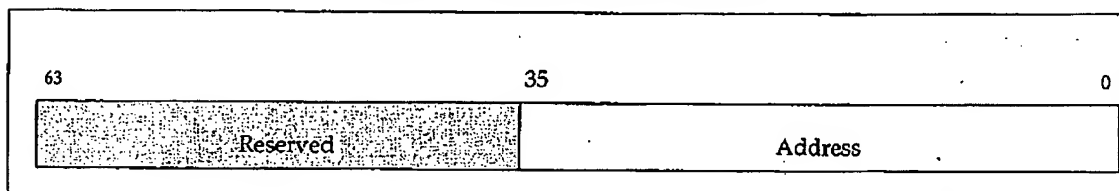
General

Each of the Machine Check Architecture register banks consists of all or a subset of the following registers (where i = the number of the register bank):

- **MCi_CTL.** Control register for error logging register bank i . Each of the 64 bits in this register, referred to as EE[63:0], are used to selectively enable/disable the generation of the machine check exception for a specific error condition related to the processor unit(s) associated with register bank i . Setting a bit enables the error reporting for a specific error condition, while clearing the bit disables it. Currently, Intel does not document the error condition related to each bit and also specifies that the programmer is only permitted to write all zeros or all ones into this register. In other words, there are only two choices: enable all errors for the bank or disable all of them.
- **MCi_STATUS.** Status register for error logging register bank i . When register bank i logs an error associated with its respective processor unit(s), the error is logged in this register. For a detailed description of the MCi_STATUS register, refer to "MCi_STATUS Register" on page 424.
- **MCi_ADDR.** See Figure 22-6 on page 424. Address register for error logging register bank i . If the MCi_STATUS[ADDRV] (address valid) bit is set to one, this register contains the address of the instruction or the data item that is related to the error. If MCi_STATUS[ADDRV] bit is cleared to zero, there is no address recorded in this register and the register must not be read from. The address returned is either the 32-bit virtual, 32-bit linear, or 36-bit physical address, depending on the type of error encountered. *The author does not know what Intel means by "32-bit virtual." The term "virtual" is not used in this way anywhere else in the three-volume set of Intel Pentium Pro data books.*
- **MCi_MISC.** Miscellaneous information register for error logging register bank i . If the MCi_STATUS[MISCV] bit is set to one, this register contains additional information related to the error code. *This register is not implemented in any of the Pentium Pro processor's register banks. Do not read this register if it's not present in the processor.*

Pentium Pro Processor System Architecture

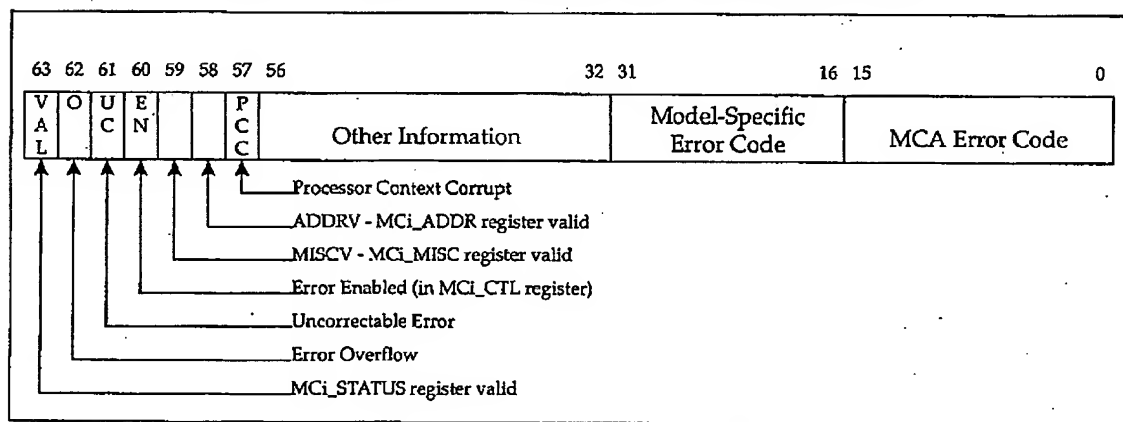
Figure 22-6: MCI_ADDR Register



MCI_STATUS Register

As described in the previous section, the MCI_STATUS register is used to record an error code related to the unit(s) associated with this error logging register bank. The register is pictured in Figure 22-7 on page 424 and the bit fields are described in Table 22-1 on page 425. Prior to examining the other bits fields within the register, the programmer should first consult the MCI_STATUS[VAL] bit (1 = error logged) to determine whether or not the register has logged a valid error.

Figure 22-7: MCI_STATUS Register for Error Logging Bank i



Chapter 22: Machine Check Architecture

Table 22-1: MCi_STATUS Register Bit Assignment

Bit Field Name	Description
VAL	Valid bit. This bit is set to one when an error is logged in the MCi_STATUS register for error logging bank <i>i</i> . If another error code has been logged previously (VAL was already set to 1), the processor uses the rules stated in the description of the Overflow bit (see next row) in deciding whether or not to overwrite the previously-logged error if another error is detected before the programmer clear this register. The programmer is responsible for clearing this bit.
O	<p>Overflow bit. When set, indicates that another Machine Check error occurred while a previously-logged error was still present in the status register. The processor sets this bit, but software is responsible for clearing it. The processor uses the following criteria in deciding whether or not to overwrite the previously-logged error with the new one:</p> <ul style="list-style-type: none"> • Enabled errors (respective MCi_CTL EE bit = 1) overwrite disabled errors (respective MCi_CTL EE bit = 0). In other words, the assumption is made that the programmer cares more about error conditions that the programmer had previously-enabled than those that the programmer had left disabled. • Uncorrected errors overwrite corrected errors. Errors that the processor cannot automatically correct are consider more important than those that were detected and automatically corrected. • A second uncorrected error does not overwrite a previously-logged uncorrected error. Uncorrected errors are very important and will not be discarded.
UC	Uncorrected. When set, indicates that the processor did not correct or wasn't able to correct the error. When clear, the processor was able to clear the error condition (e.g., a corrected ECC error when reading from an internal cache).
EN	Enabled. When set, indicates that the programmer had enabled reporting of the error condition by setting the respective bit in the error logging bank's MCi_CTL register.

Pentium Pro Processor System Architecture

Table 22-1: MCi_STATUS Register Bit Assignment (Continued)

Bit Field Name	Description
MISCV	MCi_MISC register valid. When set, the MCi_MISC register contains additional information related to the error condition. When clear, it does not.
ADDRV	MCi_ADDR register valid. When set, the MCi_ADDR register contains an address related to the error condition. When clear, it does not.
PCC	Processor Context Corrupt. When set, indicates that the error left the processor in a state where it would be dangerous to resume execution of the program that was interrupted by the Machine Check exception. When clear, the error did not affect the processor's ability to reliably resume execution of the interrupted program.
Other Information	The meaning of the bits within this field is processor implementation-specific and aren't defined by the Machine Check Architecture specification. Only software that is written for a specific processor implementation can properly interpret this field.
Model-Specific Error Code	As the name implies, these error codes are processor model-specific and may differ among various x86 processors. Different processors may use different model-specific error codes for the same condition.
MCA Error Code	Machine Check Architecture-defined error code. These error codes are strictly defined by the specification and are guaranteed to be consistent across all future x86 processors that implement the Machine Check Architecture. For a detailed description of these error codes, refer to "Machine Check Architecture Error Format" on page 429.

MSR Addresses of the Machine Check Registers

The Machine Check registers are located at the MSR addresses indicated in Table 22-2 on page 427.

Chapter 22: Machine Check Architecture

Table 22-2: MSR Addresses of the Machine Check Registers

Machine Check Register	MSR Address	Notes
MCG_CAP	179h, 377d	Intel doesn't say that these addresses will remain the same for all x86 processors, but it is the author's opinion that they will (for consistency).
MCG_STATUS	17Ah, 378d	
MCG_CTL	17Bh, 379d	
MC0_CTL	400h, 1024d	This register is aliased to the EBL_CR_POWERON MSR (see Table 3-3 on page 46). Intel states that, for this reason, only platform-specific software (typically, the BIOS) should write to this register.
MC0_STATUS	401h, 1025d	
MC0_ADDR	402h, 1026d	
MC0_MISC	403h, 1027d	Not implemented in current versions of the processor.
MC1_CTL	404h, 1028d	
MC1_STATUS	405h, 1029d	
MC1_ADDR	406h, 1030d	
MC1_MISC	407h, 1031d	Not implemented in current versions of the processor.
MC2_CTL	408h, 1032d	
MC2_STATUS	409h, 1033d	
MC2_ADDR	40Ah, 1034d	
MC2_MISC	40Bh, 1035d	Not implemented in current versions of the processor.

Pentium Pro Processor System Architecture

Table 22-2: MSR Addresses of the Machine Check Registers (Continued)

Machine Check Register	MSR Address	Notes
MC3_CTL	40Ch, 1036d	Please note that the Intel documentation shows addresses 410h through 413h for bank 3 and 40Ch through 40Fh for bank 4. The author has reversed these because he believes that the banks are implemented in sequential address order.
MC3_STATUS	40Dh, 1037d	
MC3_ADDR	40Eh, 1038d	
MC3_MISC	40Fh, 1039d	Not implemented in current versions of the processor.
MC4_CTL	410h, 1040d	See note for bank 3.
MC4_STATUS	411h, 1041d	
MC4_ADDR	412h, 1042d	
MC4_MISC	413h, 1043d	Not implemented in current versions of the processor.

Initialization of Register Set

Intel recommends that the following code sequence be used to initialize the Machine Check Architecture registers to a clean state at startup time:

```

Execute CPUID instruction to obtain features support information
Test EDX[14] and EDX[7] to determine if Machine Check exception and register set implemented
IF processor supports Machine Check exception then
    IF processor supports Machine Check Architecture register set
        IF MCG_CAP[MCG_CTL_P] = 1 (indicating that MCG_CTL present)
            Set MCG_CTL register to all ones (enable all error reporting features)
        Set variable COUNT = MCG_CAP[COUNT] field
        FOR i = 1 through COUNT DO (start at bank 1; don't write to EBL_CR_POWERON)
            Set MCi_CTL register to all ones (enable all errors for bank i)
        For i = 0 through COUNT DO
            Clear MCi_STATUS to all zeros (clear any errors)
        Set CR4[MCE] = 1 (enable Machine Check exception)

```

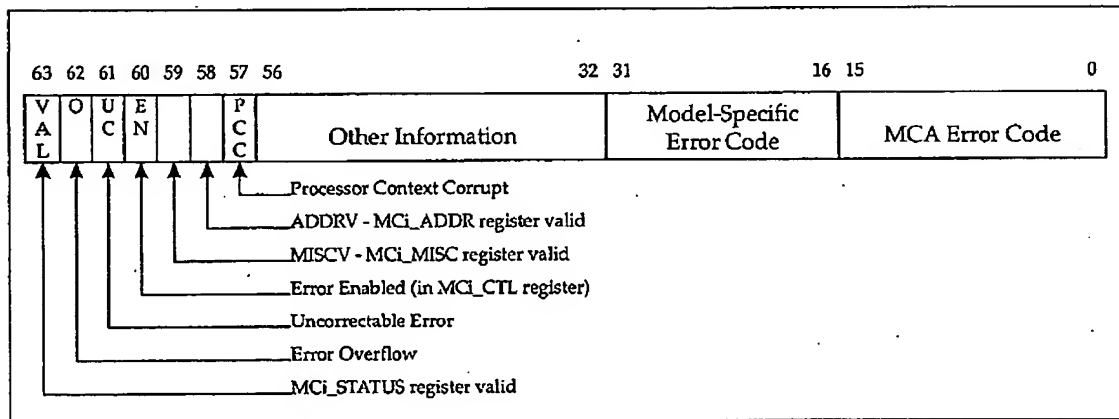
Chapter 22: Machine Check Architecture

Machine Check Architecture Error Format

As illustrated in Figure 22-8 on page 429, the lower 16-bits of the `MCi_STATUS` register contains the error code defined by the Machine Check Architecture specification, guaranteed to be consistent across all future x86 processors that implement the Machine Check Architecture registers. These error codes have two forms:

- **Simple error codes.** These error codes are generic in nature and are described in the section entitled “Simple Error Codes” on page 429.
- **Compound error codes.** These error codes describe errors related to the TLBs, memory, caches, the external bus, and the buses that interconnect processor units. They are described in the section entitled “Compound Error Codes” on page 430.

Figure 22-8: `MCi_STATUS` Register Bit Assignment



Simple Error Codes

The simple error codes defined by the architecture specification are listed in Table 22-3 on page 430.

Pentium Pro Processor System Architecture

Table 22-3: Simple Error Codes

Error	Error Value	Description
No error	0000h	No error has been logged in this register bank.
Unclassified	0001h	This error has not been classified into the architecture-defined error classes.
Microcode ROM Parity error	0002h	Intel does not specify which microcode ROM. It could be either the ROM that contains the BIST (built-in self-test) code or the ROM that contains the micro-op instructions. In either case, this is a very serious error.
External error	0003h	This processor detected BINIT# asserted by another processor, causing this processor to generate a Machine Check exception.
FRC error	0004h	The slave processor of an FRC processor pair observed an error in a transaction performed by the master processor.
Internal unclassified	0000 01xx xxxx xxxxb	Internal unclassified errors.

Compound Error Codes

As stated earlier, these error codes describe errors related to the TLBs, memory, caches, the external bus, and the buses that interconnect processor units. They take one of the three forms listed in Table 22-4 on page 431. As shown in this table, each error code form contains two or more named bit fields. Table 22-5 on page 431 through Table 22-8 on page 432 define the meaning of these bit fields.

The standard error code name defined by the architecture specification is constructed by substituting the literal text selected by the value of the respective bit field for the brace-delineated bit field within the template. The text within the template that is not enclosed in braces (including the underscore) is used as is (i.e., no substitution). As an example, when applied to the appropriate template

Chapter 22: Machine Check Architecture

from Table 22-4 on page 431, the error code value of 0000 0001 0001 0001b yields the error code name ICACHEL1_RD_ERR, indicating a level 1 instruction cache read error.

Table 22-4: Forms of Compound Error Codes

Error Type	Form (in binary)	Template
TLB errors	0000 0000 0001 TTLL	{TT}TLB{LL}_ERR
Memory hierarchy errors	0000 0001 RRRR TTLL	{TT}CACHE{LL}_{RRRR}_ERR
Bus and interconnect errors	0000 1PPT RRRR IILL	BUS{LL}_{PP}_{RRRR}_{II}_ERR Please note that the definition in the Intel manual actually reads: BUS{LL}_{PP}_{RRRR}_{II}_{TIMEOUT}_ERR but the author thinks this is incorrect and should be shown as stated above.

Table 22-5: Transaction Type Sub-Field (TT)

Transaction Type	Literal Text	Binary Value
Instruction	I	00b
Data	D	01b
Generic	G	10b

Table 22-6: Memory Hierarchy Sub-Field (LL)

Hierarchy Level	Literal Text	Binary Value
Level 0	L0	00b
Level 1	L1	01b
Level 2	L2	10b
Generic	LG	11b

Pentium Pro Processor System Architecture

Table 22-7: Request Sub-Field (RRRR)

Request Type	Literal Text	Binary Value
Generic error	ERR	0000b
Generic read	RD	0001b
Generic write	WR	0010b
Data read	DRD	0011b
Data write	DWR	0100b
Instruction fetch	IRD	0101b
Prefetch	PREFETCH	0110b
Eviction	EVICT	0111b
Snoop	SNOOP	1000b

Table 22-8: Definition of the PP, T, and II Fields

Sub-Field	Transaction	Literal Text	Binary Value
PP (Participation)	Local processor originated request	SRC	00b
	Local processor responded to request	RES	01b
	Local processor observed error as 3rd party	OBS	10b
	Generic		11b
T (Timeout)	Request timed out	TIMOUT	1
	Request did not time out		0

Chapter 22: Machine Check Architecture

Table 22-8: Definition of the PP, T, and II Fields (Continued)

Sub-Field	Transaction	Literal Text	Binary Value
II (Memory or IO)	Memory access	M	00b
	Reserved		01b
	IO access	IO	10b
	Other transaction		11b

External Bus Error Interpretation

Table 22-9 on page 433 provides detailed information on the content of the architecture-defined error code, model-specific error code, and other information fields of the MCI_STATUS register (see Figure 22-7 on page 424) for an external bus-related error.

Table 22-9: MCI_STATUS Breakdown for Bus-related Errors

Bit	Field	Description
1:0	MCA Error Code	Undefined
3:2	"	Bit 2 = 1, then special transaction. Bit 3 = 1, then special transaction or IO transaction.
7:4	"	00WR, where W = 1 indicates write and R = 1 indicates read
9:8	"	Undefined
10	"	= 0 for all EBL (external bus logic) errors. = 1 for internal watchdog timer timeout. In this case, all other bits in the MCA error field are cleared to 0. A watchdog timer timeout only occurs if BINIT# driver enabled.
11	"	= 1 for EBL errors. = 0 for internal watchdog timer timeout.
15:12	"	Reserved
18:16	Model-Specific Error Code	Reserved

Pentium Pro Processor System Architecture

Table 22-9: MCI_STATUS Breakdown for Bus-related Errors (Continued)

Bit	Field	Description
24:19	"	<p>Although Intel doesn't define the following terms, the author believes that they have the following meaning:</p> <ul style="list-style-type: none"> • BQ = Bus Queue (IOQ) • DCU = Data Cache Unit • IFU = Instruction Fetch Unit. <p>Where possible, the author has speculated on the meaning of the following error code fields.</p> <ul style="list-style-type: none"> • 000000 = BQ_DCU_READ_TYPE error. Attempted read transaction. • 000010 = BQ_IFU_DEMAND_TYPE error. Attempted read line transaction generated by the Instruction Fetch Unit due to a miss on the instruction cache. • 000011 = BQ_IFU_DEMAND_NC_TYPE error. Attempted single-quadword read from non-cacheable memory generated by the instruction fetcher. • 000100 = BQ_DCU_RFO_TYPE error. Request For Ownership stands for read and invalidate transaction. • 000101 = BQ_DCU_RFO_LOCK_TYPE error. Locked read and invalidate transaction. • 000110 = BQ_DCU_ITOM_TYPE error. Attempted read with intent to modify transaction (i.e., read and invalidate). May differ from the previous two entries by virtue of being for 0 bytes. • 001000 = BQ_DCU_WB_TYPE error. Attempted transaction to write a modified line back to memory. • 001010 = BQ_DCU_WCEVICT_TYPE error. Attempted transaction to write the contents of one of the processor's write-combining buffers back to memory. In this case, all 32 bytes don't have to be written, so performed as a series of single-quadword transfers. • 001011 = BQ_WCLINE_TYPE error. Attempted write line transaction to write the full 32 bytes in a write-combining buffer to memory. • 001100 = BQ_DCU_BTM_TYPE error. Branch Trace Message error. • 001101 = BQ_DCU_INTACK_TYPE error. Attempted Interrupt Acknowledge transaction. • 001110 = BQ_DCU_INVAL2_TYPE error. Attempted to invalidate one or more entries in the L2 cache. • 001111 = BQ_DCU_FLUSH2_TYPE error. Attempted to flush the L2 cache. • 010000 = BQ_DCU_PART_RD_TYPE error. Attempted to read a quadword (or a subset of a quadword). • 010010 = BQ_DCU_PART_WR_TYPE error. Attempted to write a quadword (or a subset of a quadword). • 010100 = BQ_DCU_SPEC_CYC_TYPE error. Attempted Special Transaction. • 011000 = BQ_DCU_IO_RD_TYPE error. Attempted IO read. • 011001 = BQ_DCU_IO_WR_TYPE error. Attempted IO write. • 011100 = BQ_DCU_LOCK_RD_TYPE error. Attempted locked memory read. • 011101 = BQ_DCU_LOCK_WR_TYPE error. Attempted locked memory write. • 011110 = BQ_DCU_SPLOCK_RD_TYPE error. Attempted locked memory read with Split Lock asserted.
27:25	"	<ul style="list-style-type: none"> • 000 = BQ_ERR_HARD_TYPE error. Transaction resulted in hard failure. • 001 = BQ_ERR_DOUBLE_TYPE error. ECC double-bit error detected on data read. • 010 = BQ_ERR_AERR2_TYPE error. Address parity error on 1st and 2nd transaction attempt. • 100 = BQ_ERR_SINGLE_TYPE error. ECC single-bit failure. • 101 = BQ_ERR_AERR1_TYPE error. Address parity error on 1st transaction attempt.
28	"	1 = FRCERR active.

Chapter 22: Machine Check Architecture

Table 22-9: MCI_STATUS Breakdown for Bus-related Errors (Continued)

Bit	Field	Description
29	"	1 = BERR# asserted by this processor.
30	"	1 = BINIT# asserted by this processor.
31	"	Reserved
34:32	Other Info	Reserved
35	"	BINIT# sampled asserted.
36	"	Processor received a parity error on a response received from another agent.
37	"	Processor received a hard fail response from another agent.
38	"	<p>ROB timeout. No micro-op has been retired for a pre-determined amount of time. Occurs when the 15-bit ROB Timeout Counter has a carry out of its high-order bit. The timer is cleared under the following circumstances:</p> <ul style="list-style-type: none"> • When a micro-op retires. • When an exception is detected. • When RESET# is asserted. • When a ROB BINIT occurs. <p>Each time that the 8-bit PIC timer has a carry, the ROB Timeout Counter is incremented by one. The PIC timer divides the bus clock by 128.</p>
41:39	"	Reserved
42	"	This processor initiated a transaction that received a Hard Fail response.
43	"	The processor has experienced a failure that caused it to assert IERR#.
44	"	This processor initiated a transaction that received an AERR# during the request phase. Upon retrying the transaction, it again received an AERR# in the request phase.
45	"	Uncorrectable ECC error. Syndrome field is in bits 54:47.
46	"	Correctable ECC error. Syndrome field is in bits 54:47.
54:47	"	Contains the 8-bit ECC syndrome only if a correctable or uncorrectable ECC error occurred and there wasn't already a previous valid ECC syndrome in this field (indicated by bit 45 = 1). After processing an ECC error, the software must clear bit 45 to permit the logging of future ECC syndromes.
56:55	"	Reserved

23

Performance Monitoring and Timestamp

The Previous Chapter

The previous chapter described the error logging and reporting Machine Check Architecture first introduced in the Pentium processor and greatly expanded in the Pentium Pro processor. The Machine Check Architecture defines an exception (the Machine Check exception) used to report hardware-related problems and a register set used to log both recoverable and unrecoverable errors.

This Chapter

This chapter describes the performance monitoring and time stamp counter facilities incorporated in the Pentium Pro processor. It should be noted that these facilities are also implemented in the Pentium processor, but Intel did not document them until the release of the Pentium Pro processor.

The Next Chapter

The next chapter provides an introduction to the MMX instruction set. It should be noted that, as of this writing, MMX is not available on the Pentium Pro processor. This chapter has been included because it is well known that Intel intends to proliferate MMX throughout its processor product lines, including future versions of the Pentium Pro processor.

Pentium Pro Processor System Architecture

Time Stamp Counter Facility

Time Stamp Counter (TSC) Definition

Basically, the Time Stamp Counter (TSC) is a very accurate elapsed time measurement tool. It is a 64-bit counter that counts the number of processor clock cycles that have occurred since reset was removed (or since the programmer cleared the counter). As an example, on a processor whose internal clock speed is 200MHz, the TSC has a resolution of 5ns.

The TSC is incremented even during periods when the processor has halted or when the STPCLK# input has been asserted, causing the processor to disable the clocking of its internal units. The TSC, the local APIC, and the external bus interface still receive the clock.

When the timer reaches a counter of all ones, it wraps around to all zeros and continues counting. No interrupt is generated as a result of the wraparound. Intel guarantees that the minimum time that it takes the TSC to go from a count of zero to a count of all ones will never (on future x86 processors) be less than 10 years. For the Pentium and Pentium Pro processors, the wraparound period will take several thousands of years (in other words, the current processors are pretty darn slow compared to their future cousins).

Detecting Presence of the TSC

A processor's support for the TSC counter is detected by executing the CUID instruction (see "CUID Instruction Enhanced" on page 359) with a request for the feature support information. If EDX[4] = 1, the processor supports the TSC.

Accessing the Time Stamp Counter

Reading the TSC Using RDTSC Instruction

The programmer may read the TSC using the RDTSC (read TSC) instruction. For more information, refer to "Read Time Stamp Counter (RDTSC)" on page 370. Please note that the RDTSC is not serializing and may therefore not yield an accurate elapsed time. For more information, refer to "RDTSC Doesn't Serialize" on page 370.

Chapter 23: Performance Monitoring and Timestamp

Reading the TSC Using RDMSR Instruction

On the Pentium and the Pentium Pro processors, the TSC may also be read using the RDMSR instruction (see “Accessing MSRs” on page 371). The MSR address of the TSC is 10h (16d). Note that the ability to read the TSC using the RDMSR instruction is processor-specific and may not be implemented in future processors.

Writing to the TSC

The TSC can be written to using the WRMSR instruction (see “Accessing MSRs” on page 371). The MSR address of the TSC is 10h (16d). Only the lower 32-bits of the 64-bit TSC can be written to, however. All zeros are written throughout the upper 32-bits of the TSC when the lower half of the register is written to.

Performance Monitoring Facility

Purpose of the Performance Monitoring Facility

The performance monitoring facility is invaluable when tuning, or profiling, program code to yield the best possible performance. It permits the programmer to obtain an accurate profile of how efficiently the processor and memory are utilized by a program. The Pentium and Pentium Pro processors each include two performance monitoring counters that can be programmed independently of each other to take one of two types of measurements:

- measure the duration of a specific event type.
- measure the number of occurrences of a specific event type.

The event types that can be measured are processor-dependent and include a large number of types such as cache hits, cache misses, TLB hits and misses, etc.

Performance Monitoring Registers

The performance monitoring facility is implemented using four MSRs:

- The performance event select MSRs, `PerfEvtSel0` and `PerfEvtSel1`. `PerfEvtSel0` is used to control performance counter 0 (`PerfCtr0`), while `PerfEvtSel1` is used to control performance counter 1 (`PerfCtr1`).
- The performance counter MSRs, `PerfCtr0` and `PerfCtr1`.

These registers are described in the sections that follow.

Pentium Pro Processor System Architecture

PerfEvtSel0 and PerfEvtSel1 MSRs

The performance event select registers are used to set up and enable or disable performance counters 0 and 1. They both have the format illustrated in Figure 23-1 on page 441. Table 23-1 on page 440 provides a detailed description of each bit field.

Table 23-1: PerfEvtSel0 and PerfEvtSel1 MSR Bit Assignment

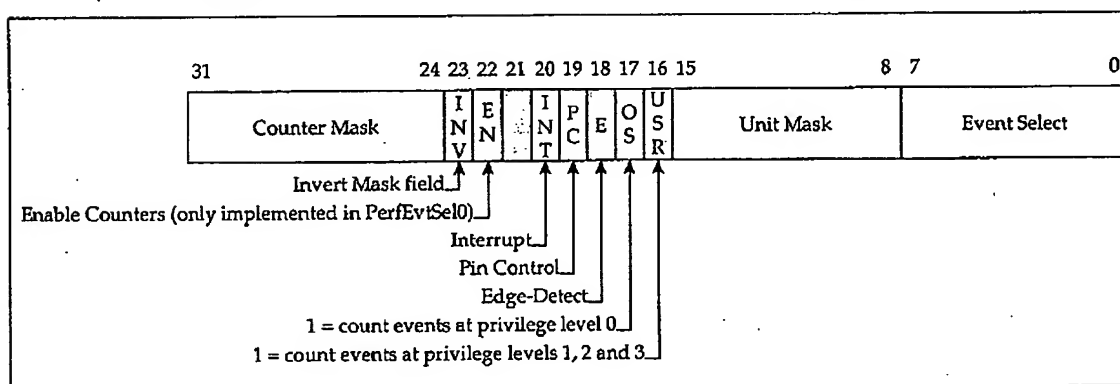
Field Name	Description
Counter Mask	When cleared to zero and the respective event counter has been enabled, the counter increments once for each event of the selected type. When set to a non-zero value, the value is used as a threshold trigger point. When the number of events of the specified type is \geq this value, the event counter is incremented by one.
INV	Invert bit. When set, the processor inverts the result of the counter mask comparison so that less than (i.e., $<$) comparisons can be made.
EN	Enable bit. This bit is only implemented in PerfEvtSel0. When it is set to one, both counters are enabled. When cleared to zero, both counters are disabled.
INT	Interrupt Enable bit. When set to one, the local APIC is enabled to generate an interrupt when the respective counter overflows.
PC	Pin Control bit. Assuming that the PC bit = 0, the processor toggles the PB_n (where n = counter number) output pin each time that the respective counter increments. When the PC bit = 1, the processor toggles the PB_n output pin when the respective counter overflows. A toggle of the PB_n output pin is defined as its assertion for two BCLKs followed by its deassertion.
E	Edge Detect bit. When set, enables the detection of deasserted to asserted events of any condition that can be specified by the other fields. Note that it cannot recognize back-to-back assertions.

Chapter 23: Performance Monitoring and Timestamp

Table 23-1: PerfEvtSel0 and PerfEvtSel1 MSR Bit Assignment (Continued)

Field Name	Description
OS	Operating System bit. OS = 1 and USR = 0—the respective counter only counts events that occur while the processor is executing privilege level 0 code. OS and USR = 1—the respective counter counts events that occur while the processor is executing code at any privilege level. OS = 0 and USR = 1—the respective counter only counts events that occur while the processor is executing privilege level 1, 2, or 3 code.
USR	User bit. See the description of the OS bit.
Unit Mask	The unit mask field usage is event-specific and further qualifies the event type.
Event Select	This field is used to select the type of event to be monitored.

Figure 23-1: PerfEvtSel0 and PerfEvtSel1 MSR Bit Assignment



PerfCtr0 and PerfCtr1

Each of the performance counters is 40-bits wide and is independently-controlled by its respective PerfEvtSel register. For a detailed description of how to read or write the counter registers, refer to the section entitled "Accessing the PerfCtr MSRs" on page 442.

Pentium Pro Processor System Architecture

Accessing the Performance Monitoring Registers

Accessing the PerfEvtSel MSRs

The PerfEvtSel MSRs are accessed using the RDMSR and WRMSR instructions. These instructions can only be executed successfully (i.e., without causing a GP exception) while the processor is executing a privilege level 0 program or when executing in real mode. These two registers are implemented at the following MSR addresses:

- PerfEvtSel0 MSR is located at MSR address 186h (390d).
- PerfEvtSel1 MSR is located at MSR address 187h (391d).

Accessing the PerfCtr MSRs

Accessing Using RDPMC Instruction. The RDPMC (read performance counter) instruction, governed by the state of CR4[PCE], can be used to read the contents of the specified counter:

- When CR4[PCE] = 0, the RDPMC instruction can only be successfully executed (without causing a GP exception) by programs executing at privilege level 0.
- When CR4[PCE] = 1, the RDPMC instruction can successfully be executed by programs executing at any privilege level.

Accessing Using RDMSR/WRMSR Instructions. The PerfCtr MSRs can only be read and written directly using the RDMSR and WRMSR instructions (at privilege level 0). They are implemented at the following MSR addresses:

- PerfCtr0 is located at MSR address C1h (193d).
- PerfCtr1 is located at MSR address C2h (194d).

The 40-bit performance counters can be written to using the WRMSR instruction. However, the programmer can only directly write to the lower 32 bits of the target counter. The high-order bit is duplicated in the high-order eight bits of the counter. This allows the programmer to initialize the counters with both positive (bit 31 = 0) and negative (bit 31 = 1) values.

Chapter 23: Performance Monitoring and Timestamp

Event Types

For a detailed description of each event type, refer to appendix B in the Intel data book entitled *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Guide*.

Starting and Stopping the Counters

Starting the Counters

The performance counters are started as follows:

1. write valid setup information into the respective PerfEvtSel register using the WRMSR instruction.
2. the counter begins counting after the successful execution of a WRMSR instruction that sets the EN (enable counters) bit in the PerfEvtSel0 MSR.

Stopping the Counters

Both counters may be stopped by:

- clearing the EN (enable counters) bit in the PerfEvtSel0 MSR, or
- clearing all bits in the PerfEvtSel MSRs.

Just counter 1 may be stopped by writing all zeros to the PerfEvtSel1 MSR.

Performance Monitoring Interrupt on Overflow

The following steps must be taken when enabling a counter to generate an interrupt on counter overflow:

- A counter is enabled to generate an interrupt when the counter overflows by setting the INT bit in its respective PerfEvtSel MSR to one.
- The programmer must set up the local APIC's PCINT (performance counter interrupt) entry in its LVT (local vector table). For more information, refer to "Added APIC Functionality" on page 402.
- The programmer initializes the IDT (interrupt descriptor table) entry specified in the local APIC's PCINT LVT entry to point to the performance counter overflow interrupt handler routine.

24

MMX: Matrix Math Extensions

The Previous Chapter

The previous chapter described the performance monitoring and time stamp counter facilities incorporated in the Pentium Pro processor. It should be noted that these facilities are also implemented in the Pentium processor, but Intel did not document them until the release of the Pentium Pro processor.

This Chapter

This chapter concludes the software part of the book. It provides an introduction to the MMX instruction set. It should be noted that, as of this writing, MMX is not available on the Pentium Pro processor. This chapter is included because it is well known that Intel intends to proliferate MMX throughout its processor product lines, including future versions of the Pentium Pro processor.

Please Note

This chapter is not intended to be a comprehensive reference to the MMX instruction set. Rather, it is intended to serve as a brief introduction to the concept of MMX. Intel has a comprehensive and clearly written MMX document set available on their web site.

Problems Addressed by MMX

Problem: Math on Packed Bytes/Words/Dwords

Consider the following example scenario—the programmer must add two video images in memory together to yield a resultant video image. In the example, each pixel of each of the images is represented by a separate byte in mem-

Pentium Pro Processor System Architecture

ory. In order to perform the calculation, the programmer must take the following steps:

1. Read the first byte from the first image in memory.
2. Perform the calculation with the first byte in the second image in memory.
3. Store the resultant byte into the first byte of the area of memory that will contain the resultant image.
4. Repeat steps 1 through 3 until the third image has been produced.

If it is assumed that the video subsystem is in 1024 x 768 mode, each screen image consists of 786432 pixels, or bytes. That's how many times steps 1 through 3 would have to be repeated. This will consume a tremendous amount of the processor's clock cycles and generates 786432 x 2 memory reads (one to read 1st byte from 1st image and 1 to get 1st byte from the 2nd image) and 786432 memory writes to update the 3rd image.

Solution: MMX Matrix Math/Logical Operations

The MMX extensions include a set of 8 MMX registers, each 64-bits wide. Using the appropriate MMX instructions and the MMX register set, the operation described in the previous section could be accomplished as follows (see Figure 24-1 on page 447):

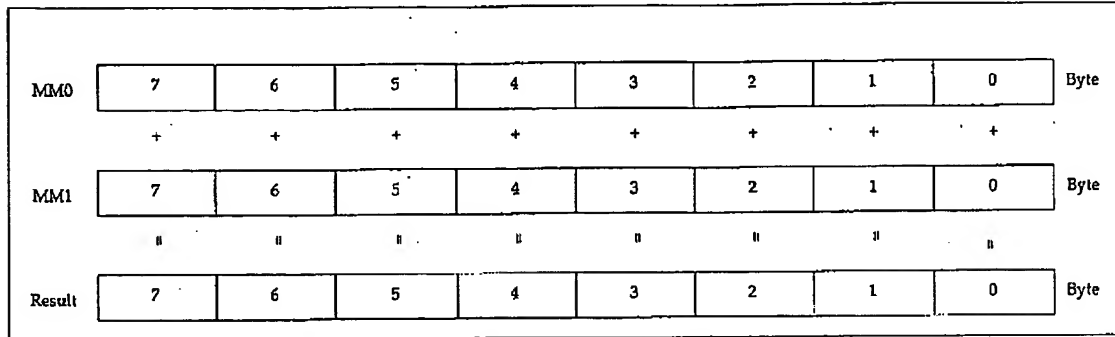
1. Read the 1st 8 bytes from the 1st image in memory into one of the MMX registers. The MMX register then contains a packed array (or matrix) of 8 pixel bytes).
2. Read the 1st 8 bytes from the 2nd image in memory into another of the MMX registers. The MMX register then contains a packed array (or matrix) of 8 pixel bytes).
3. Execute a matrix instruction that adds the two registers together, treating bytes in the same relative position in the two MMX registers as a separate add operation. In other words, the MMX add packed-bytes instruction uses 8 separate adders to add the two registers together. The bytes that result from the 8 independent adds are stored in their respective positions in the target MMX register.
4. Perform 1 write to write the 8 packed bytes into the first 8 locations of the third image in memory.
5. Repeat steps 1 through 4 until the third image has been produced.

This operation results in only 12.5% of the activity that was generated using non-MMX operations. The MMX instruction set includes a separate version of each instruction to perform math and logical operations on packed bytes,

Chapter 24: MMX: Matrix Math Extensions

packed words, packed dwords, or quadwords (unpacked data). Assuming the packed byte example, the processor can conclude the task 8 times more rapidly than before and generate only 12.5% of the traffic. The processor is more available to handle other tasks.

Figure 24-1: Example MMX Packed-Byte Add



Problem: Data not Packed

The previous example assumed that the data to be operated upon already resided in memory in packed byte, word, or dword form. This is not always the case and it requires a lot of software overhead to assemble unpacked data in memory into packed form in an MMX register in preparation for a matrix math or logical operation. Likewise, after the matrix operation has been completed, the reverse operation must be performed—store the data from the MMX register back into memory in its original, unpacked form.

Solution: MMX Pack and Unpack Instructions

The MMX instruction set includes instructions that read unpacked data from memory and pack it into an MMX register. It also includes instructions that write packed data in an MMX register back to memory in unpacked form.

Problem: Math Overflows/Underflows

In the previous example, two video images were added together to produce a third, resultant image. The example, however, ignored the possibility of overflows when each pair of pixel bytes were added together. Normally, when performing calculations on pixel information, software must handle the resultant

Pentium Pro Processor System Architecture

overflow and underflow conditions. This is necessary to prevent pixels from ending up the wrong color or intensity, and adds considerably to the software overhead.

Solution: Saturating Math

The MMX instruction set includes versions of each calculation type that perform saturated math: if the operation results in an overflow or underflow, the resultant value is clamped to its maximum or minimum value, respectively. This eliminates the costly software overhead normally associated with this type of operation.

Problem: Comparisons and Branches

Everyone has seen the weatherperson standing in front of the weather map. The image of the person is taken in front of a plain, blue background. The image containing the person is then combined with the map image, replacing all blue pixels in the first image with the corresponding map pixels.

Assume that each of the two images consists of 16-bits per pixel (24 million colors per pixel) stored in memory as a series of packed 16-bit pixels. The image containing the person is referred to as image X and the map image as image Y.

Without MMX code, each of the 16-bit pixels would have to be handled as follows:

```
cmp    x[i], BLUE    ;check if pixel is blue
jne    next_pixel    ;if not, process next pixel
mov     x[i], y[i]    ;pixel=blue, so replace with respective map pixel
```

This code fragment would have to be executed for every pixel in the image. Not only is this code-intensive, but it involves a conditional branch based on random data (it is not predictable which pixels are blue versus those that aren't). The chances of mispredictions are high, and we know that mispredicted branches result in costly performance degradation in the Pentium Pro processor.

Solution: MMX Parallel Comparisons

The code in the previous example can be replaced with the following MMX code fragment:

Chapter 24: MMX: Matrix Math Extensions

```

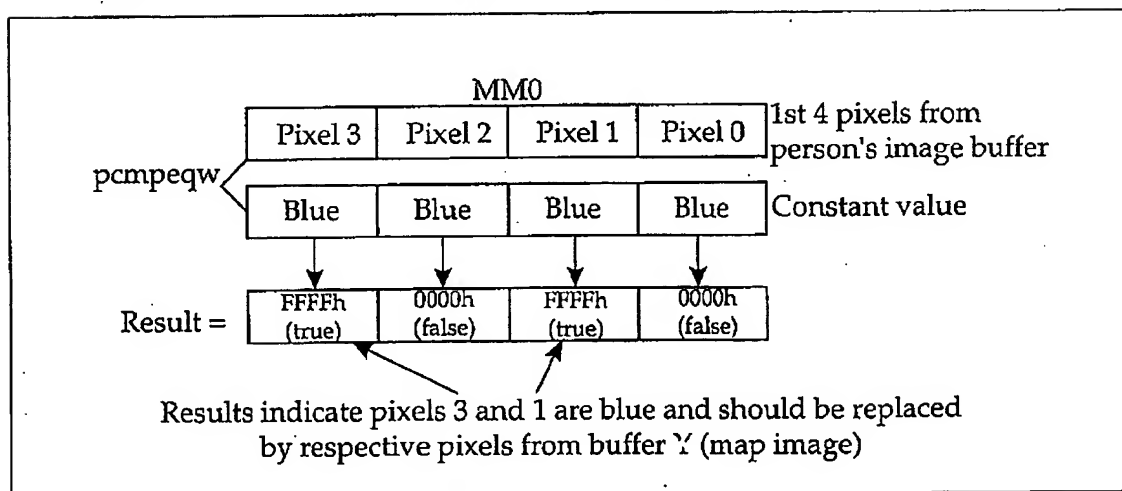
mov  mm0, x[i]      ;get 4 packed pixels from person's image in mm0
pcmpeqwmm0, BLUE    ;compare to identify unwanted pixels
pandn x[i], mm0      ;keep person pixels, 0 blue ones in buffer x
pand  mm0, y[i]      ;keep map pixels, 0 positions to place person pixels
por   x[i], mm0      ;combine map with person in buffer x

```

1. The first instruction (`mov`) fetches 4, 16-bit packed pixels from the image with the person on the blue background.
2. The second instruction (`pcmpeqw`) performs a packed word compare for words that are equal to a constant containing packed BLUE pixel values. The result is placed in MM0 (see Figure 24-2 on page 449).
3. The third instruction (`pandn`) performs a packed negative AND operation using the mask values created by instruction 2 on the person's image in buffer X in memory (see Figure 24-3 on page 450). The result is stored in buffer X.
4. The fourth instruction (`pand`) performs a packed AND operation using the mask values created by instruction 2 on the map image in buffer Y in memory (see Figure 24-4 on page 450). The result overwrites the mask value in MM0.
5. The fifth instruction (`por`) performs a packed OR operation to combine the four pixels of the person with the four pixels of the map (see Figure 24-5 on page 451).

This routine will be faster (it simultaneously operates on two arrays of four pixels each) and also eliminates the conditional branches and therefore the possibility of mispredicts.

Figure 24-2: Results of PCMPEQW (Packed Compare If Words Equal) Instruction



Pentium Pro Processor System Architecture

Figure 24-3: Results of PANDN (Packed Logical AND NOT) Instruction

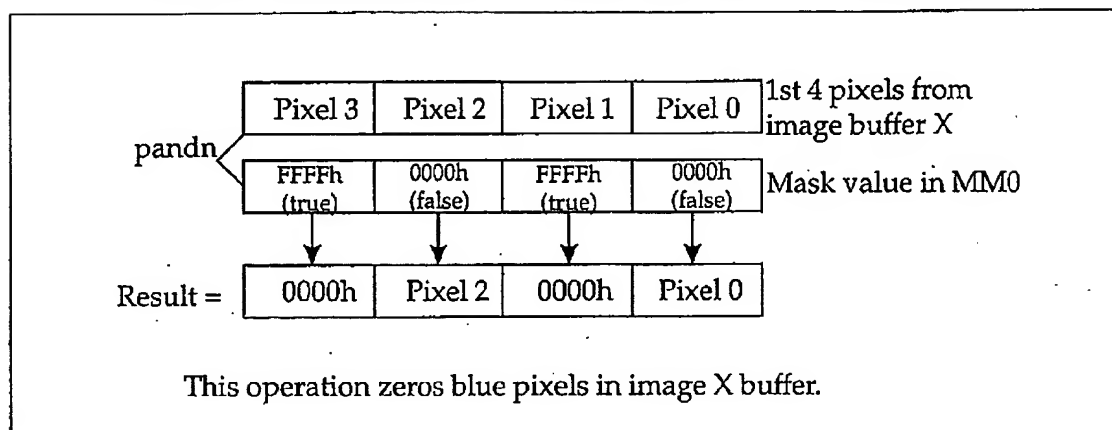
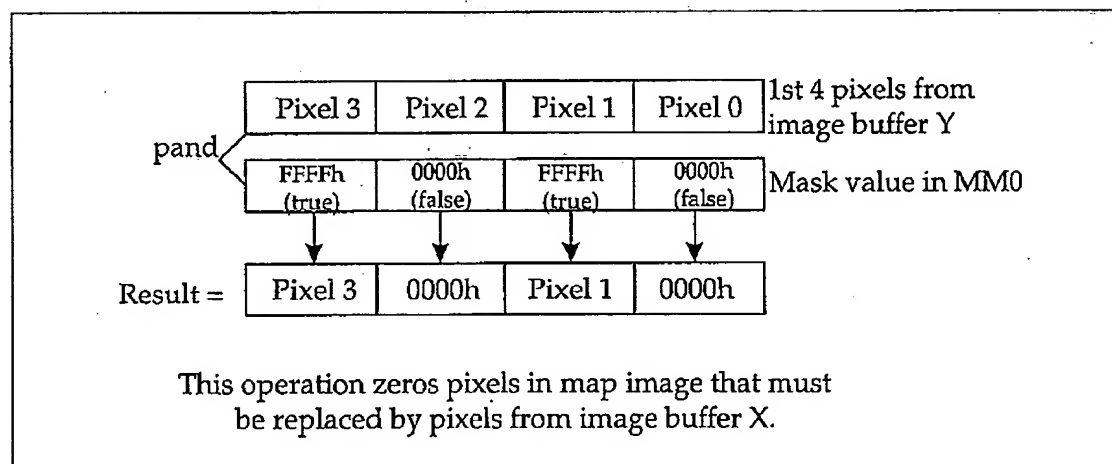
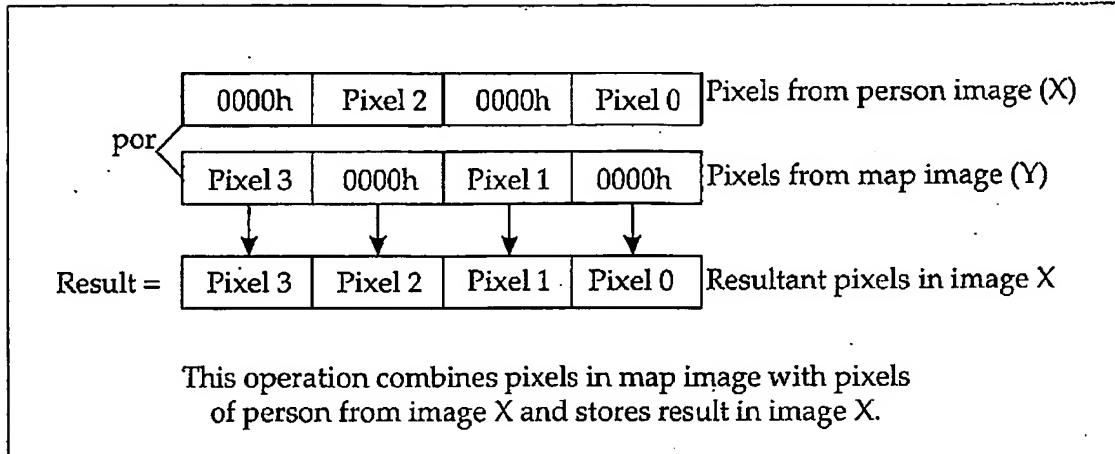


Figure 24-4: Result of PAND (Packed Logical AND) Instruction



Chapter 24: MMX: Matrix Math Extensions

Figure 24-5: Result of POR (Packed Logical OR) Instruction



Single Instruction, Multiple Data (SIMD)

Intel uses the term SIMD to describe the fundamental premise underlying the MMX instruction set: a single instruction capable of operating on multiple, packed data items.

Detecting Presence of MMX

The MMX extensions can be detected by executing the CPUID instruction with a feature request (see "CPUID Instruction Enhanced" on page 359). EDX[23] = 1 indicates that the processor supports MMX.

Changes to Programming Environment

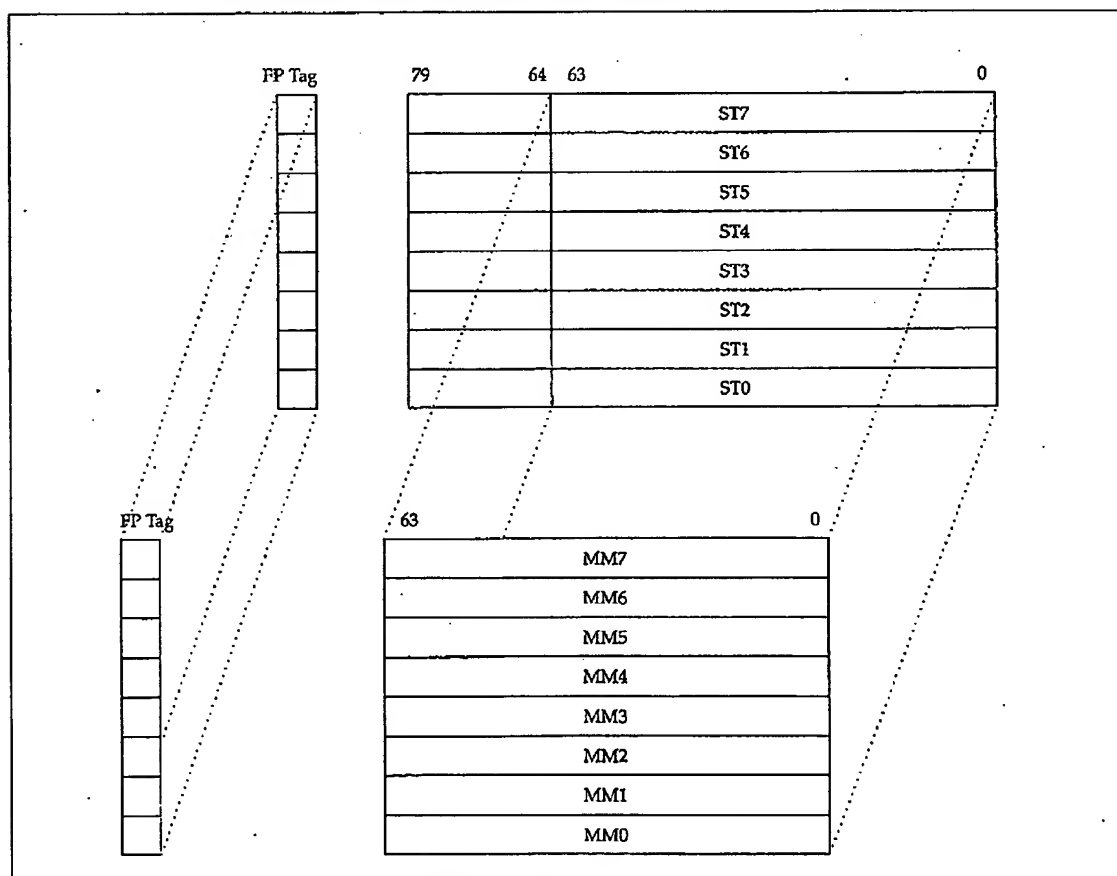
General

In a word: none. No new exceptions, modes, interrupts, etc. There are 8 new MMX registers, MM[7:0], each 64-bits wide, but Intel took pains to ensure that the new registers didn't add to the number of registers that would have to be saved and reloaded on a task switch—the MMX registers are mapped right over the FPU registers (see Figure 24-6 on page 452) which are already saved and reloaded on a task switch (see "Handling a Task Switch" on page 453).

Pentium Pro Processor System Architecture

- The MM[7:0] registers are aliased on the 64-bit mantissa portion of the FP registers.
- When a value is written to one of the MMX registers, it also appears in the mantissa portion of the respective FP register. The reverse is also true—a value written to a FP register also appears in the respective MMX register. In other words, the lower 64-bits of each FP register has two separate names that it responds to—the MMX register name and the FP register name.
- When a value is written to an MMX register, bits [79:64] of the corresponding FP register are all set to ones.
- The FP registers are addressable as stack locations. Pushes and pops are used to put a value onto the FP register that is currently at the TOS (top-of-stack) position and pops are used to read the value from the FP register that is currently at the TOS position. The MMX registers, on the other hand, are each explicitly addressed by name.

Figure 24-6: MMX Registers are Mapped Over FP Registers



Chapter 24: MMX: Matrix Math Extensions

Handling a Task Switch

When a task switch occurs, the processor saves most of its register set contents in the current task's TSS (Task State Segment data structure) and automatically sets `CR0[TS] = 1` as it enters the next task. However, the processor doesn't automatically save the state of the FPU registers on a task switch (the TSS data structure that a task's register image is stored in doesn't contain entries for the FPU registers).

It is possible that the task just suspended was using the FPU or the MMX registers, and they must be saved before the current task starts using the FPU or MMX registers. If the `CR0[TS]` (task switch) bit is set and a FP or MMX instruction is executed, this results in a Device Not Available exception (`INT 7`). The OS's `INT 7` handler can use the same code that it uses to save the FPU state to save the MMX state. Both the `FSAVE` and `FRSTOR` instructions are used to save and restore either the FP or MMX state, whichever register set is currently in use. After the `INT 7` handler saves the FPU or MMX register set and clears the `CR0[TS]` bit, the handler can then execute a return instruction, causing the processor to reattempt execution of the FPU or MMX instruction. The instruction now executes successfully (because `CR0[TS]` is no longer set).

When Exiting MMX Routine, Execute EMMS

As stated earlier, the MMX register set consists of 8 registers, `MM[7:0]`, each 64-bits in width. They are mapped onto the FPU register set, `FP[7:0]`. The moment that the first MMX instruction is executed, two things occur:

- the MMX registers take the place of the FPU registers.
- the FPU tag word is marked valid.

Because the FPU tag word is marked valid (indicating that the FPU registers contain valid data—something that is not true), it is imperative that an EMMS (Empty MMX State) instruction be executed after completion of the MMX code and before any FPU code is executed.

MMX Instruction Set

Instruction Groups

The MMX instruction set is divided into the following groups:

Pentium Pro Processor System Architecture

- Arithmetic instructions
- Comparison instructions
- Conversion instructions
- Logical instructions
- Shift instructions
- Data transfer instructions
- Empty MMX State (EMMS) instruction

Instruction Syntax

Each MMX instruction is built from the following elements:

- Optional P (prefix) indicating that it deals with packed data.
- Instruction operation—e.g., ADD, CMP, or XOR.
- Optional data-type Suffix:
 - US—Unsigned Saturation.
 - S—signed saturation.
 - B—packed byte.
 - W—packed word.
 - D—packed dword.
 - Q—quadword.

As an example, the PADDUSW instruction mnemonic represents a packed add using unsigned saturated addition operating on packed words.

Some instructions that have different input and output elements have two data-type suffixes. As an example, the conversion instruction converts from one data type to another and has two suffixes: one for the original data type and one for the converted data type.

Instruction Set

Table 24-1 on page 455 defines the MMX instruction set.

Chapter 24: MMX: Matrix Math Extensions

Table 24-1: MMX Instruction Set

Instruction Group	Mnemonic	Description
Data Transfer, Pack, Unpack	MOV[D,Q]	Move dword or quadword to/from MMX register.
	PACKUSWB	Pack words into bytes with unsigned saturation.
	PACKSS[WB,DW]	Pack words into bytes, or dwords into words, with signed saturation.
	PUNPCKH[BW,WD,DQ]	Unpack (interleave) high-order bytes, words, or dwords from MMX register.
	PUNPCKL[BW,WD,DQ]	Unpack (interleave) low-order bytes, words, or dwords from MMX register.
Arithmetic	PADD[B,W,D]	Packed add on bytes, words, or dwords.
	PADDS[B,W]	Saturating add on bytes or words.
	PADDUS[B,W]	Unsigned saturated add on bytes or words.
	PSUB[B,W,D]	Packed subtract on bytes, words, or dwords.
	PSUBS[B,W]	Saturating subtraction on bytes or words.
	PSUBUS[B,W]	Unsigned saturating subtract on bytes or words.
	PMULHW	Multiply packed words to get high bits of product.
	PMULLW	Multiply packed words to get low bits of product.
	PMADDWD	Multiply packed words to get pairs of products.
Shift and rotate	PSLL[W,D,Q]	Packed shift left logical on words, dwords, or quadwords.
	PSRL[W,D,Q]	Packed shift right logical on words, dwords, or quadwords.
	PSRA[W,D]	Packed shift right arithmetic on words or dwords.
Logical	PAND	Bit-wise logical AND.
	PANDN	Bit-wise logical AND NOT.
	POR	Bit-wise logical OR.
	PXOR	Bit-wise logical XOR.

Pentium Pro Processor System Architecture

Table 24-1: MMX Instruction Set (Continued)

Instruction Group	Mnemonic	Description
Compare	PCMPEQ[B,W,D]	Packed compare if equal on bytes, words, or dwords.
	PCMPGT[B,W,D]	Packed compare if greater than on bytes, words, or dwords.
Miscellaneous	EMMS	Empty MMX state.

Pentium Pro MMX Execution Units

As of this writing, MMX has not yet been implemented on the deliverable versions of the Pentium Pro processor. However, Intel has information posted on their web site that provide insight into how it will be implemented. The MMX execution units are connected to the Reservation Station (RS) as illustrated in Figure 24-7 on page 457:

- Port 0: MMX ALU Unit and MMX Multiplier Unit
- Port 1: MMX ALU Unit and MMX Shifter Unit.

The MMX execution units have the following characteristics:

- The Reservation Station can simultaneously (i.e., in the same clock) dispatch MMX instructions through ports 0 and 1.
- Since there is an MMX ALU unit on each of these two ports, two MMX ALU instructions can be executed simultaneously.
- MMX multiply and shift instructions can simultaneously be dispatched and begin execution.

Part 4: Overview of Intel Pentium Pro Chipsets

The Previous Part

Part 3 provided a description of the enhancements to the software environment.

This Part

Part 4 provides an overview of the Intel 450KX, 450GX, and 440FX chipsets (the chipsets that had been released as of this writing). It consists of the following chapters:

- "450GX and KX Chipsets" on page 461.
- "440FX Chipset" on page 497.

25

450GX and KX Chipsets

This Chapter

This chapter provides an overview of the Intel 450GX and 450KX Pentium Pro chipsets.

The Next Chapter

The next chapter provides an overview of the Intel 440FX Pentium Pro chipset.

Processor Bus Operation

For a detailed description of the Pentium Pro processor bus operation, refer to:

- "Hardware Section 2: Bus Intro and Arbitration" on page 177
- "Hardware Section 3: The Transaction Phases" on page 239
- "Hardware Section 4: Other Bus Topics" on page 305

PCI Bus Operation

For a detailed description of PCI bus operation, refer to the MindShare book entitled *PCI System Architecture* (published by Addison-Wesley).

450GX Chipset

Overview

Refer to Figure 25-1 on page 464. The 450GX chipset supports up to a total of eight devices on the processor bus:

Pentium Pro Processor System Architecture

- a cluster of up to four processors.
- one or two host/PCI bridges. They are referred to as the compatibility and the auxiliary PBs (PCI bridges).
- one or two memory controllers, each capable of controlling up to 4GB of memory. Each memory controller consists of a DC (DRAM controller), a DP (data path unit), and four MICs (memory interface components). Collectively, they place one load on the processor bus.

As a variation, Figure 25-2 on page 465 illustrates two processor clusters interconnected by a cluster bridge. This model still meets the limit of eight devices on the processor bus, even if each processor bus each incorporates two PBs.

Major Features

The 450GX chipset incorporates the following major features:

- Supports 60 or 66MHz processor bus speed and 30 or 33MHz PCI bus speed.
- In addition to performing parity checking on the address and the REQ[4:0]# signal groups, ECC checking on the data bus is supported.
- Compatibility PB supports the boot ROM and the ISA bus beyond the bridge. In other words, it acts as the response agent for processor-initiated accesses to ISA devices and to the boot ROM.
- The two PBs share usage of the BPRI# signal when PCI masters require access to main memory. Only one device may assert BPRI# at a time, however. In the event that both PBs require access to main memory simultaneously, the compatibility PB acts as the arbiter between the two. The two sideband signals IOREQ# and IOGNT# are used by the aux PB to request ownership of the BPRI# signal.
- The main memory controller can be configured to handle either 1-way, 2-way, or 4-way interleaved memory, up to a maximum of 4GB in size.
- If there are two memory controllers implemented, they can be configured to occupy contiguous or non-contiguous memory ranges.
- The memory controller and the PBs are highly configurable regarding the memory and/or IO ranges that they recognize when acting as the target of a transaction originated on either side of the PBs.
- The memory controller can be programmed to control access to System Management memory.
- Both the PBs and the memory controller can be programmed with different accessibility attributes relative to the address ranges that they are configured to recognize.
- When acting as the target of processor-initiated transactions that target PCI

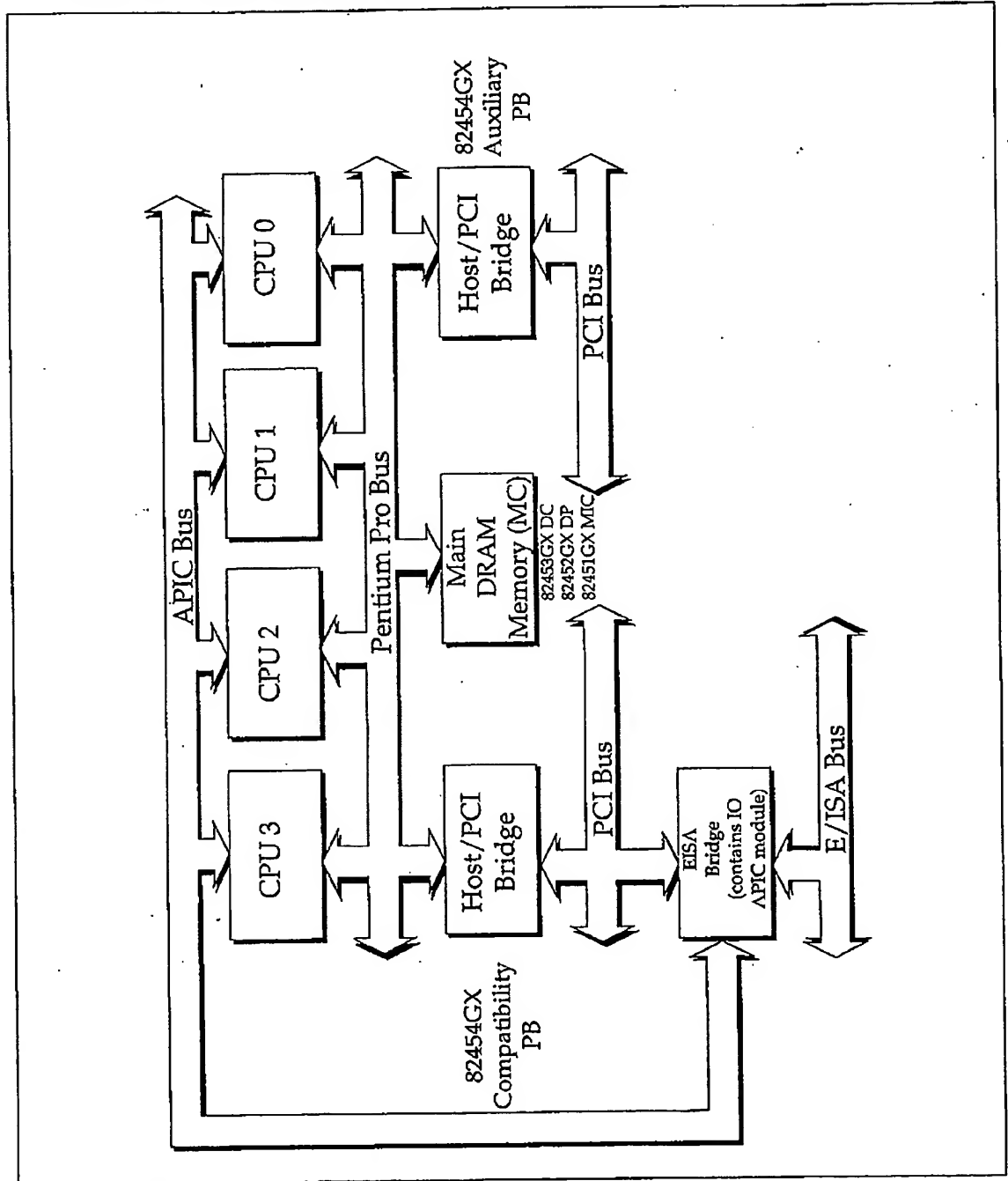
Chapter 25: 450GX and KX Chipsets

devices, each PB can accept up to four transactions in its outbound transaction queue.

- Each PB can accept up to four inbound PCI-to-main memory transactions in its inbound transaction queue.
- Each memory controller can handle up to four transactions that target main memory.
- Each PB contains four, 32-byte outbound data buffers that handle processor-to-PCI memory writes and PCI reads from main memory.
- Each PB contains four, 32-byte inbound data buffers that handle PCI-to-main memory writes and processor-initiated reads from PCI memory.
- The memory controller contains four, 32-byte outbound data buffers that handle writes to main memory.
- The memory controller contains four, 32-byte inbound data buffers that handle reads from main memory.

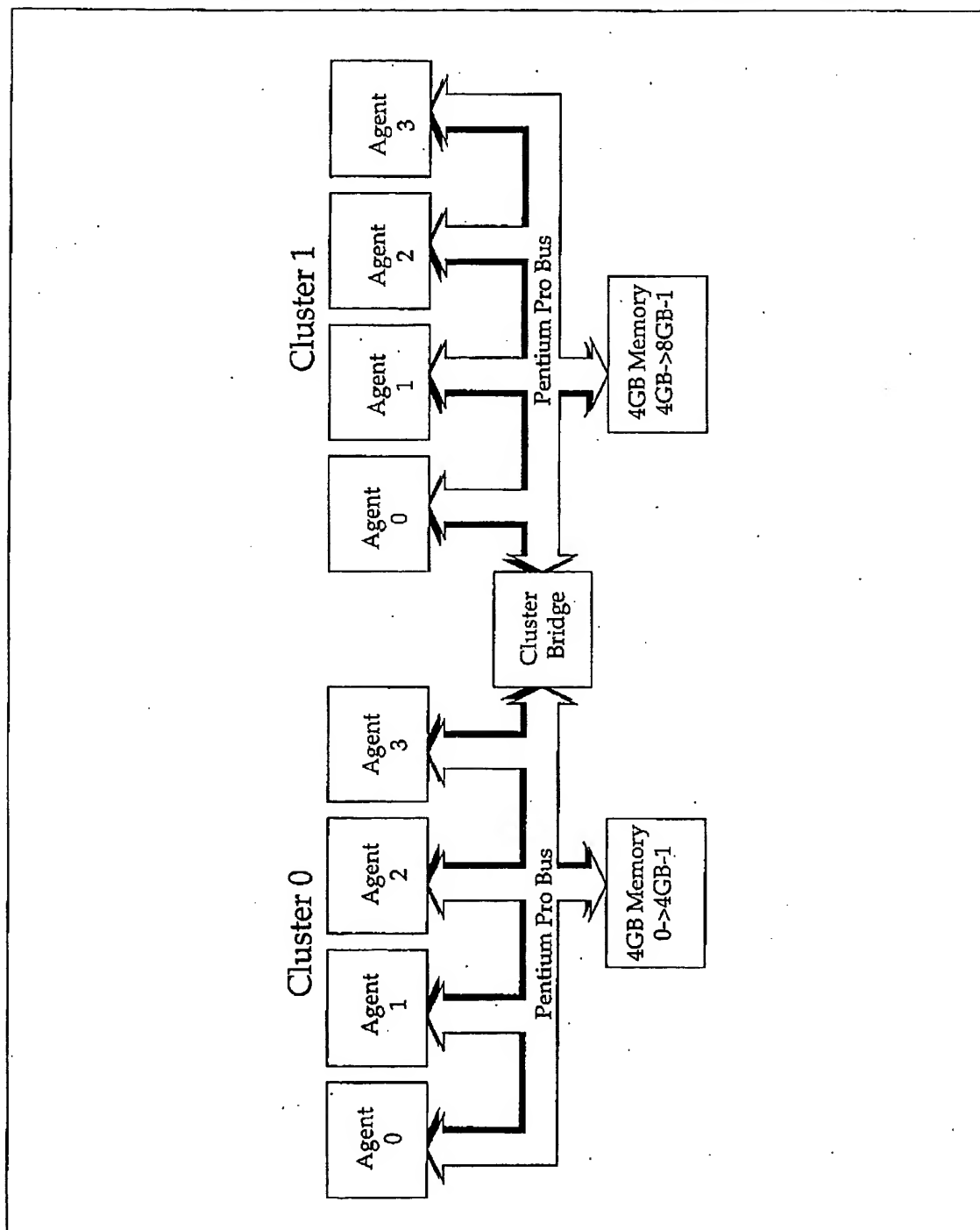
Pentium Pro Processor System Architecture

Figure 25-1: Block Diagram of Typical System Designed Using Intel 450GX Chipset



Chapter 25: 450GX and KX Chipsets

Figure 25-2: Two Pentium Pro Clusters Connected by a Cluster Bridge



Pentium Pro Processor System Architecture

Overview of Compatibility PB

The compatibility PB resides between the processor bus and a PCI bus. In addition, the PCI-to-ISA or PCI-to-EISA bridge resides beyond the compatibility PB. When a transaction is initiated on either side of the compatibility PB, the compatibility PB, along with all of the other devices on that bus, latches the transaction information. It then has to make one of three decisions:

1. The compatibility PB itself is the target.
2. The target device is on the other side of the compatibility PB.
3. Neither the compatibility PB nor a device on the other side is the target of the transaction.

Compatibility PB is the Target

In other words, the transaction initiator is addressing the compatibility PB itself. In this case, the compatibility PB doesn't pass the transaction through to the bus on the other side. Rather, it acts as the target of the transaction. The next two sections describe the two possible cases.

Transaction Initiated on Processor Bus. In the following cases, the compatibility PB acts as the response agent for the transaction, but does not pass the transaction through to the PCI bus:

- Processor has initiated a special transaction that issues a **flush, sync, flush acknowledge, or SMI acknowledge message**.
- Processor has initiated a **branch trace message** transaction. The PB acts as the response agent, asserting TRDY# to indicate when it's ready to accept the data. It does not accept the data, however. A debug tool monitors for the branch trace message transaction and accepts (using snarfing) the data when it sees the processor assert DRDY# to indicate the presence of the data.
- Processor has initiated an **IO write to the configuration data port, the configuration address port, or the TRC register**.
- Processor has initiated an **IO read from the configuration data port** (to access one of the compatibility PB's configuration registers) or the TRC register.

Transaction Initiated on PCI Bus. There are no cases where a transaction initiated on the PCI bus is targeting the compatibility PB itself. There are no IO ports within the PB that are accessible from the PCI side and the PB has no internal memory that can be targeted.

Chapter 25: 450GX and KX Chipsets

Target on Other Side of Compatibility PB

The transaction initiator is addressing a device that resides on the other side of the compatibility PB. In this case, the PB must arbitrate for ownership of the bus on the other side and must re-issue the transaction on that side. Since the target resides on the opposite side of the PB, the PB must act as the surrogate for the target on the other side. The next two sections describe the two cases:

Transaction Initiated on Processor Bus. The compatibility PB acts as the response agent for the following transactions and must pass the transaction through to the PCI bus:

- Processor-initiated **interrupt acknowledge** transactions are passed onto the PCI bus as a PCI interrupt acknowledge transactions. When the ISA bridge returns the interrupt vector, it is passed back to the processor.
- Processor-initiated special transactions that issue the **shutdown, halt, or stop grant acknowledge message** are passed to the PCI bus as a PCI special cycle transaction. The message is broadcast on the PCI bus during the PCI transaction's data phase.
- Processor-initiated **IO read or write** that targets a PCI or an EISA device, or that addresses an IO port that falls within the range that may be used by an ISA device (IO addresses 0100h through 03FFh or any address that aliases to this range).
- Processor-initiated **memory read or write** that targets a PCI or an EISA device, or that addresses memory that falls within the range that may be used by an ISA device (memory addresses 00000000h through 000FFFFFFh).

Transaction Initiated on PCI Bus. The only PCI-initiated transactions that are passed through to the processor bus are memory transactions that target main memory. The PB acts as the target of the PCI transaction and acts as the request agent on the processor bus.

Neither Compatibility PB nor Device on Other Side Is Target

In this case, the compatibility PB ignores the transaction.

Overview of Aux PB

The aux PB resides between the processor bus and a PCI bus. Unlike the compatibility PB, however, the PCI-to-ISA or PCI-to-EISA bridge doesn't reside beyond the aux PB. When a transaction is initiated on either side of the aux PB,

Pentium Pro Processor System Architecture

the aux PB, along with all of the other devices on that bus, latches the transaction information. It then has to make one of three decisions:

1. The aux PB itself is the target.
2. The target device is on the other side of the aux PB.
3. Neither the aux PB nor a device on the other side is the target of the transaction.

Aux PB is the Target

In other words, the transaction initiator is addressing the aux PB itself. In this case, the aux PB doesn't pass the transaction through to the bus on the other side. Rather, it acts as the target of the transaction. The next two sections describe the two possible cases.

Transaction Initiated on Processor Bus. There is only one case where the aux PB acts as the response agent of a processor-initiated transaction where the aux PB itself is the target of the transaction. This is the case where the processor is accessing the aux PB's configuration registers through the configuration data port.

Transaction Initiated on PCI Bus. There are no cases where a transaction initiated on the PCI bus is targeting the aux PB itself. There are no IO ports within the PB that are accessible from the PCI side and the PB has no internal memory that can be targeted.

Target on Other Side of Aux PB

The transaction initiator is addressing a device that resides on the other side of the aux PB. In this case, the PB must arbitrate for ownership of the bus on the other side and must re-issue the transaction on that side. Since the target resides on the opposite side of the PB, the PB must act as the surrogate for the target on the other side. The next two sections describe the two cases:

Transaction Initiated on Processor Bus. The compatibility PB acts as the response agent for the following transactions and must pass the transaction through to the PCI bus:

- Processor-initiated IO read or write that targets a PCI device.
- Processor-initiated memory read or write that targets a PCI device.

Transaction Initiated on PCI Bus. The only PCI-initiated transactions that are passed through to the processor bus are memory transactions that tar-

Chapter 25: 450GX and KX Chipsets

get main memory. The PB acts as the target of the PCI transaction and acts as the request agent on the processor bus.

Neither Aux PB nor Device on Other Side Is Target

In this case, the aux PB ignores the transaction.

Overview of Memory Controller

The memory controller (MC) consists of the following components:

- a DC (DRAM controller)
- a DP (data path unit)
- four MICs (memory interface components).

Collectively, they place one load on the processor bus. When a transaction is initiated on the processor bus, the memory controller latches the transaction along with all other bus agents. It must then make one of four decisions:

1. Main memory is the target.
2. SMM memory is the target.
3. The memory controller's configuration registers are the target.
4. The transaction has nothing to do with main memory or SMM memory.

In the first three cases, the memory controller acts as the response agent in the transaction. In the fourth case, it ignores the transaction.

Startup Autoconfiguration

Introduction

When the platform is first powered up, the PBs sample input pins to determine what their agent IDs are and what their PCI configuration device numbers are.

Autoconfiguration of PBs

On the rising-edge of the PWRGD signal, both of the PBs sample their IOREQ# and IOGNT# pins to determine which one is the compatibility PB and which is the aux PB. Table 25-1 on page 470 shows the resultant bridge type and ID assignments based on the values sampled from the pins. Note the following:

Pentium Pro Processor System Architecture

- The PBs require an agent ID assigned because they initiate transactions on the processor bus and a request agent is required to supply its agent ID during the transmission of request packet B.
- The PBs' configuration registers are accessed using the same mechanism used to access the configuration registers associated with PCI devices. For more information, refer to "How Chipset Members are Configured by Software" on page 476.

As a result of the powerup assignment, the two PBs have the default responsibilities indicated in Table 25-2 on page 470 and Table 25-3 on page 471.

Table 25-1: PB Powerup ID Assignment

IOREQ#	IOGNT#	Bridge Type	Agent ID	PCI Configuration Device Number
0	1	Compatibility PB	9	19h (25d)
1	0	Aux PB	Ah (10d)	1Ah (26d)

Table 25-2: Default Startup Responsibilities of Compatibility PB

Responsibilities
Acts as the response agent for writes to the configuration address port.
Passes accesses for boot ROM to PCI bus (because the boot ROM is located on the ISA bus).
Turbo Reset Control (TRC) register can be accessed. For more information, see "Processor Bus Agent Configuration" on page 473.
Acts as the response agent for branch trace message (but does not latch data) and special transactions issued with the halt, stop grant, and shutdown messages.
Acts as the response agent for the special transaction issued with the SMI acknowledge message. When the first SMI acknowledge message is detected (with SMMEM# asserted in request packet B), the PB asserts the SMIACK# signal and keeps it asserted until the second SMI acknowledge message is detected (with SMMEM# deasserted in request packet B). It then deasserts SMIACK#. While asserted, SMIACK# tells the memory controller that it must enable the SMM memory decoder.
Provides the arbiter for use of BPRI# when both PBs require access to main memory.

Chapter 25: 450GX and KX Chipsets

Table 25-2: Default Startup Responsibilities of Compatibility PB (Continued)

Responsibilities
Deturbo Counter Control (DCC) register can be accessed.
IO decoders enabled, causing this PB to act as the response agent for all processor-initiated IO transactions. All IO transactions other than those directed at the configuration address and data ports and the TRC register are passed onto the PCI bus.
CONFVR register is accessible. For more information, refer to "Processor Bus Agent Configuration" on page 473.

Table 25-3: Default Startup Responsibilities of Aux PB

Responsibilities
Doesn't act as the response agent for writes to the configuration address port. Snarfs data written to it. Compatibility PB acts as the response agent.
Doesn't pass accesses for boot ROM to PCI bus (because the boot ROM is located beyond the compatibility PB on the ISA bus).
Turbo Reset Control (TRC) register cannot be accessed. For more information, see "Processor Bus Agent Configuration" on page 473.
Ignores the branch trace message and special transactions issued with the halt, stop grant acknowledge, SMI acknowledge, and shutdown messages.
Issues IOREQ# to the compatibility PB when it requires ownership of BPRI# to access main memory. It then waits for the other PB to assert IOGNT# before it asserts BPRI# to obtain ownership of the request signal group.
Deturbo Counter Control (DCC) register cannot be accessed.
IO space range registers aren't active, causing this PB to ignore all processor-initiated IO transactions other than those directed at the configuration address and data ports.
CONFVR register isn't accessible. For more information, refer to "Processor Bus Agent Configuration" on page 473.

Pentium Pro Processor System Architecture

Autoconfiguration of Memory Controller

Refer to Table 25-4 on page 472. The memory controllers sample the state of the OMCNUM (Orion memory controller number) input on the rising-edge of the PWRGD signal to determine which is the primary memory and which the secondary memory controller. In addition, the value sampled also determines the memory controller's PCI device number. Note the following:

- The memory controller doesn't need an agent ID assigned because it never initiates transactions.
- Although the memory controller doesn't reside on the PCI bus, its configuration registers are accessed using the same mechanism used to access the configuration registers associated with PCI devices. For more information, refer to "How Chipset Members are Configured by Software" on page 476.

As a result of this powerup assignment, the two memory controllers have the default responsibilities indicated in Table 25-5 on page 472 and Table 25-6 on page 473.

Table 25-4: Memory Controller Powerup ID Assignment

OMCNUM	Memory Controller Number	PCI Device Number
0	0	14h (20d)
1	1	15h (21d)

Table 25-5: Default Startup Responsibilities of Memory Controller 0

Responsibilities
Snarfs processor IO writes performed to the configuration address port. For more information, refer to "How Chipset Members are Configured by Software" on page 476.
Base address set to 000000000h.
Responds to accesses within first 512KB of memory (000000000h through 00007FFFFh) and in the 1MB through 4MB minus one range (000100000h through 0003FFFFFh).
DRAM architecture set to non-interleaved (i.e., 1-way).

Chapter 25: 450GX and KX Chipsets

Table 25-5: Default Startup Responsibilities of Memory Controller 0 (Continued)

Responsibilities
DRAM timing set to reasonable values for a 66MHz bus and slow memory.

Table 25-6: Default Startup Responsibilities of Memory Controller 1

Responsibilities
Snarfs processor IO writes performed to the configuration address port. For more information, refer to "How Chipset Members are Configured by Software" on page 476.
Base address set to 100000000h (4GB).
Responds to accesses within first 512KB of memory above the 4GB boundary (100000000h through 10007FFFFh) and in the 3MB range from 100100000h through 1003FFFFFh.
DRAM architecture set to non-interleaved (i.e., 1-way).
DRAM timing set to reasonable values for a 66MHz bus and slow memory.

Processor Bus Agent Configuration

As described in the chapter entitled "Features that are Automatically Configured" on page 37, each processor bus agent samples a series of its pins on the trailing-edge of reset. The values sampled are used to select some of the device's basic operational characteristics.

The compatibility PB implements a register, CONFVR (configuration value register), that, in conjunction with the TRC (turbo reset control) register, permits the BIOS programmer to provide a different set of configuration settings to the bus agents after reset has already been removed. This is accomplished in the following manner:

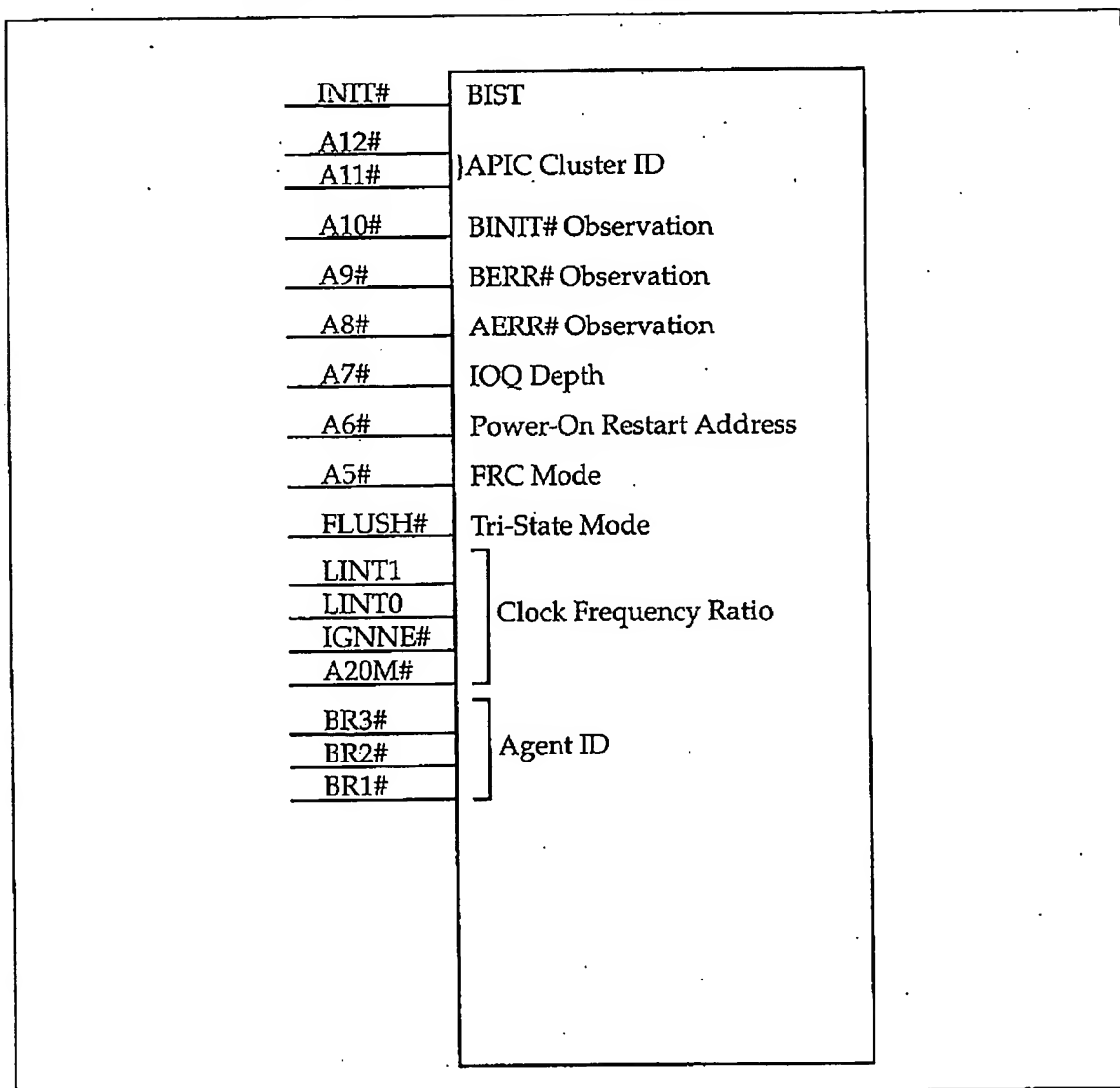
1. After program execution has been initiated, the programmer writes the desired new configuration settings to the CONFVR register.
2. The programmer then performs an IO write to the TRC register in the compatibility PB (see Figure 25-4 on page 475) with the CPU Reset bit set to one.

Pentium Pro Processor System Architecture

In response, the compatibility PB takes the following actions:

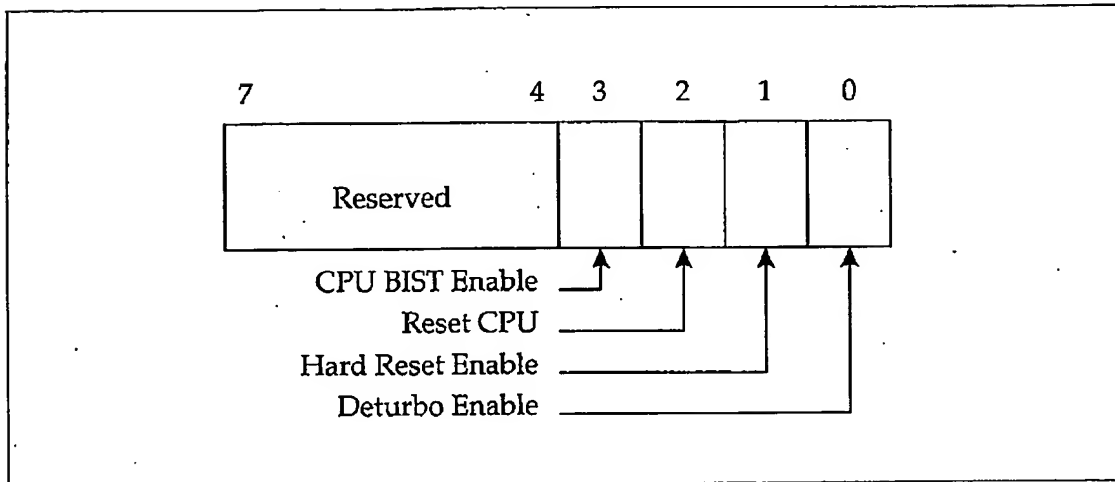
1. Asserts RESET# to all processor bus agents.
2. Drives the contents of the CONFVR register onto the signal lines specified in Figure 25-3 on page 474 that correspond to the bits in the CONFVR register.
3. Removes the reset signal.
4. After at least two BCLKs (minimum autoconfiguration signal hold time), ceases to drive the configuration signal lines.

Figure 25-3: Pins Sampled on Trailing-edge of RESET#



Chapter 25: 450GX and KX Chipsets

Figure 25-4: TRC Register Bit Assignment



Transaction Deferral Not Implemented—Retry Used Instead

The 450GX PBs do not support transaction deferral when acting either as request or response agents. Rather, when a processor initiates a transaction that targets a device on the PCI side of the bridge (i.e., either PB), the bridge takes the following actions:

1. Memorizes the transaction.
2. Asserts DEFER# in the snoop phase of the transaction (see "The Snoop Phase" on page 257).
3. Returns the retry response in the response phase of the transaction (see "Possible Solutions" on page 310).
4. If the transaction is a read, there is no data phase in the transaction.
5. If the transaction is a write, the write data is latched.
6. The processor that initiated the transaction will continually rearbitrate for the request signal group and, when it is its turn to use the bus, retry the transaction again.
7. Until the PB has completed the read or the write on the PCI side, it continually responds to the processor transaction with a retry response. The PB uses the previously-latched DID information (see "Contents of Request Packet B" on page 249) to determine that it is the same transaction.
8. When the PB has completed the PCI read or write transaction, it will respond normally to the next processor retry of the transaction (i.e., if it's a

Pentium Pro Processor System Architecture

read, it returns the read data; if it's a write, accepts the data normally, but doesn't deliver it to the PCI target because it already did it).

How Chipset Members are Configured by Software

Each of the PBs and the memory controllers implements a set of configuration registers that are programmed by the BIOS. The sections that follow define the manner in which the configuration registers are accessed and the usage of these registers.

Chipset Configuration Mechanism

The PBs and the memory controllers are configured using the configuration mechanism that is used to configure PCI devices. This may seem puzzling in the case of the memory controllers because they don't reside on the PCI bus. It's always a processor that performs device configuration for devices located on both the processor and PCI buses. The fact that the memory controllers don't truly reside on a PCI bus doesn't mean that the same method can't be used to access their configuration registers.

For a detailed description of the PCI device configuration mechanism and registers, refer to the MindShare book entitled *PCI System Architecture* (published by Addison-Wesley).

Each of the PBs and memory controllers implement the PCI configuration address port and configuration data port. Both are implemented as 32-bit IO ports. The address port occupies IO locations 0CF8h through 0CFBh (pictured in Figure 25-5 on page 478), while the data port occupies IO locations 0CFCh through 0CFFh (each of the four IO locations corresponds to a location with a dword of the selected PCI device's configuration space). To access any PCI device's configuration registers, the programmer performs the following process:

1. Perform a 32-bit IO write to the address port setting the enable bit and specifying the target PCI bus, physical PCI device, PCI function number (i.e., logical device within the physical device), and the target dword (1-of-64) within the logical device's configuration space. The only information that's missing is whether it's a configuration read or write and which bytes within the selected dword of the target device's configuration space.
2. The PBs and the memory controllers all consider themselves as residing on PCI bus 0 and each has a unique PCI physical device number that is automatically assigned to it on machine powerup (see Table 25-1 on page 470

Chapter 25: 450GX and KX Chipsets

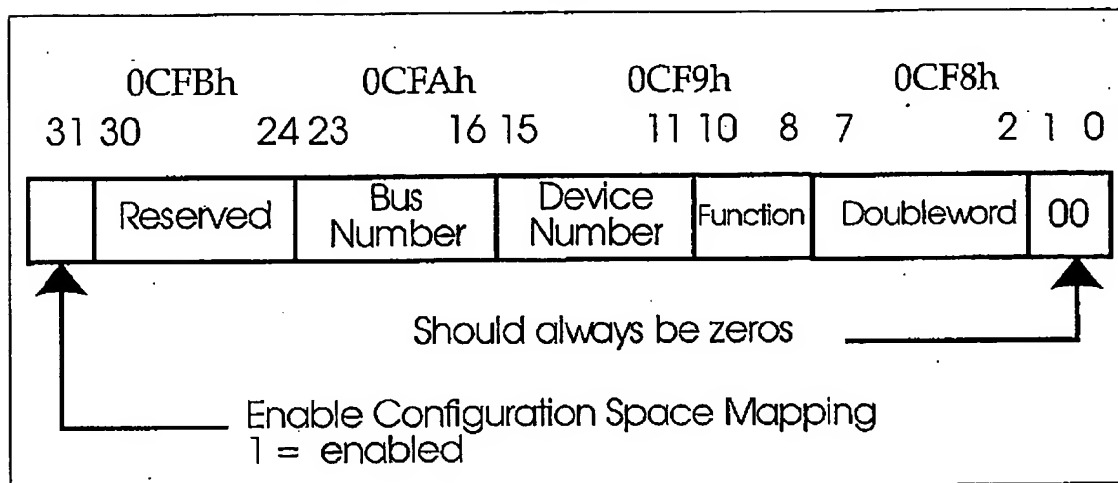
and Table 25-4 on page 472). Each of them considers itself to be a single-function PCI device with a PCI function number of 0. Remember that both PBs and both memory controllers implement the address port at the same IO locations. It would therefore seem that there will be a contention problem with possibly four devices simultaneously acting as the response agent of the transaction. Only the compatibility PB, however, acts as the response agent and actively participates in the transaction. The aux PB and the memory controllers, on the other hand, quietly snarf the data as it is accepted by the compatibility PB. At the conclusion of the IO write, therefore, both PBs and both memory controllers have accepted the write data into their address port.

3. At the conclusion of the IO write, both PBs and both memory controllers then compare the target bus, device, and function number to their own to determine which, if any of them, is the target of the soon to be performed access to the configuration data port.
4. The programmer then performs a one, two, or four byte read from or write to the configuration data port (which, once again, is implemented at the same IO locations in all of the devices). Only the device which had a match on its bus, device, function number acts as the response agent of the read or write. The other devices ignore the access to the data port. The access to the data port tells the selected device whether this is a configuration read (if it's an IO read from the data port), or a configuration write (if it's an IO write). In addition, the processor's byte enables identify which of the four locations within the selected dword are being accessed.

Now that the method for accessing a device's configuration registers has been identified, the sections that follow provide a description of the configuration registers that are implemented in the PBs and the memory controllers.

Pentium Pro Processor System Architecture

Figure 25-5: PCI Configuration Address Port



PB Configuration Registers

Figure 25-6 on page 488 illustrates the PB's configuration registers, while Table 25-7 on page 478 provides a description of each of them. This table describes the PB configuration registers for both the 450GX and 450KX PB chips.

Table 25-7: PB Configuration Registers

Offset	Description
00h-01h	Vendor ID. Intel vendor ID = 8086h
02h-03h	Device ID. PB device ID = 84C4h.
04h-05h	<p>PCI Command. Implements all PCI spec-defined bits except: Fast Back-to-Back Enable, Palette Snoop Enable, and Special Cycle Monitoring.</p> <ul style="list-style-type: none"> IO Enable bit enables/disables PCI IO accesses on host bus. Default is enabled. Memory Enable bit enables/disables PCI memory accesses on host bus. Default is enabled. Bus Master Enable bit is hardwired to the enabled state. Memory Write and Invalidate bit is disabled at startup. Parity Error Response is disabled at startup. Stepping is never used. SERR# Enable is disabled at startup. <p>For more information, refer to the MindShare book entitled "PCI System Architecture."</p>

Chapter 25: 450GX and KX Chipsets

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
06h-07h	PCI Status. Implements all PCI spec-defined bits except: 66MHz Enable. <ul style="list-style-type: none"> The bridge is not fast back-to-back capable. Status bit are all cleared at startup. Bridge is a medium-speed PCI target. For more information, refer to the MindShare book entitled "PCI System Architecture."
08h	Revision ID.
09h-0Bh	Class Code. Hardwired to value 060000h, indicating that it's a host/PCI bridge.
0Ch	Cache Line Size. Hardwired to 08h, indicating a cache line size of 32 bytes ($8 * 4 = 32$).
0Dh	Latency Timer. Default value set to 20h, indicating a maximum burst length of 32 PCI clocks. Can be changed. For more information, refer to the MindShare book entitled "PCI System Architecture."
0Eh	Header Type. Hardwired to 00h, indicating that the bridge is a single-function device with header type 0. For more information, refer to the MindShare book entitled "PCI System Architecture."
0Fh	Built-In Self-Test. Bridge does not implement a BIST. Although register is read/write, writes have no effect.
10h-3Fh	Reserved. Bridge does not implement any BARs, CardBus CIS Pointer, Subsystem Vendor ID, Subsystem ID, Expansion ROM BAR, Interrupt Line, Interrupt Pin, Min_Gnt, or Max_Lat registers.
Remainder of registers are device-specific, outside the scope of the PCI specification.	
40h-43h	Top of System Memory. If enabled, defines the top of main memory (with a granularity of 1MB). Any processor memory accesses above this address (up to the 64GB end of memory space) are forwarded to the PCI bus. Any PCI memory accesses between the top of main memory and the 64GB boundary are ignored by the bridge.
44h-47	Reserved.
48h	PCI Decode Mode. Performs two functions with regard to processor-initiated IO transactions: <ul style="list-style-type: none"> Enable/disable masking of A[31:16] before comparing IO address to the ranges defined in the two IO Space Range registers. In some cases, the processor places a one on A[16] when performing an IO transaction (referred to as IO address wraparound). This feature permits A16 to be forced to 0 by the bridge before the IO address is passed to the PCI bus. Enable/disable ISA IO aliasing. When enabled, the bridge masks A[15:10] before comparing the IO address to the ranges defined in the two IO Space Range registers. In a dual-PB GX system, both PBs must have the ISA IO Alias bit set the same. Otherwise, both may respond to the same IO address.

Pentium Pro Processor System Architecture

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
49h	<p>Bridge Device Number. This read-only register provides the bridge's PCI physical device number. Established at powerup when both PBs sample IOREQ# and IOGNT# at the leading-edge of PWRGD.</p> <ul style="list-style-type: none"> • KX, value always 19h (25d). • GX, value for compatibility PB = 19h (25d), while value for aux PB = 1Ah (26d).
4Ah	<p>PCI Bus Number. Read/write register. Number of PCI bus immediately on the other side of the PB. Default value is 00h in both PBs. Programmer must change number in aux PB before attempting to access device configuration registers for devices on PCI bus 0. For more information, refer to the MindShare book entitled "PCI System Architecture."</p>
4Bh	<p>PCI SubOrdinate Bus Number. Read/write register. Number of highest-numbered PCI bus that resides beyond the PB. Default value is 00h in both PBs. Programmer must change number in aux PB before attempting to access device configuration registers for devices on PCI bus 0. For more information, refer to the MindShare book entitled "PCI System Architecture."</p>
4Ch	<p>PB Configuration. This register controls various PB features:</p> <ul style="list-style-type: none"> • If the Host Bus Timeout feature bit is enabled in the PB Extended Error Reporting Command register, a timeout of either 1.5ms (default) or 30ms may be selected in this register. • If a PCI-to-PCI bridge is present beyond the bridge, programmer must enable (default = disabled) the Lock Atomic Reads feature bit. Any reads that cross a dword boundary are then issued as locked reads. • If the Branch Trace Message Response Enable bit is enabled (default = enabled) in the compatibility PB (in a GX system) or a KX bridge, the bridge acts as the target of the Branch Trace Message transaction (but it does not latch the address delivered on the data bus). Disable this bit if a development tool needs to act as the target. In a dual-bridge GX system, the aux PB ignores this bit. • If enabled to do so (default = enabled), bridge responds to Shutdown Special transaction by asserting INIT# to the processor, typically resulting in a reboot of the OS. In a dual-bridge GX system, the aux PB ignores this bit. • PB arbiter selection. The two PBs both use the BPRI# signal to request ownership of the Pentium Pro request bus. Since it's obvious that both can't use BPRI# simultaneously, there must be a BPRI# arbiter. Possible selections are: <i>no arbitration</i> (single-bridge system); <i>this PB provides arbiter</i> (default); or <i>other PB provides arbiter</i>. In a KX system, use default. In a GX system with only one PB, select <i>no arbitration</i>. In a dual-bridge GX system, select default for compatibility PB and <i>other PB provides arbiter</i> for aux PB.
4Dh-50h	Reserved.

Chapter 25: 450GX and KX Chipsets

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
51h	DeTurbo Counter. If enabled (via a bit in the TRC register), value in this register is compared to the value in an 8-bit free-running counter running at the host bus clock/8. While the counter value > this register, BPRI# is kept asserted by PB. BPRI# is deasserted while the value is ≤ this register. Default = 80h. Provides a draconian method of making the system appear to run slower (by keeping the processors from using the host bus for periods of time). Used in KX system. Only used in compatibility PB in GX system.
52h	Reserved.
53h	CPU R/W Control. Permits the programmer to enable/disable posting of host-to-PCI writes in the PB. Default = disabled.
54h-55h	PCI R/W Control. Controls following features: <ul style="list-style-type: none"> Whether a PCI-initiated memory read line command is passed to main memory as a line read or as a series of single-beat transfers. Additionally, a memory read line can be converted into main memory line read followed by a prefetch of 3 or more lines from main memory. Default = disabled. Whether a PCI-initiated memory read multiple command is passed to main memory as a line read or as a series of single-beat transfers. Additionally, a memory read multiple can be converted into main memory line read followed by a prefetch of 3 or more lines from main memory. Default = disabled. Whether a PCI-initiated memory read command is passed to main memory as a line read or as a single-beat transfer. Additionally, a memory read can be converted into main memory line read followed by a prefetch of 3 or more lines from main memory. Default = disabled. Whether or not back-to-back line writes by processor(s) can be converted to one, continuous PCI burst write transaction (referred to as write-combining in the PCI spec). Default = disabled. Whether or not PCI-to-main memory writes can be posted in the PB. Default = disabled.
56h	Reserved.
57h	SMM Enable. When enabled, PB does not discriminate between accesses to locations in SMRAM range and PCI memory that occupies the same range. If a processor initiates a memory read or write to SM memory (SMMEM# asserted in request packet 2) and the address is within a range also assigned to PCI memory beyond the bridge, the transaction will be passed to the PCI bus. When disabled (default) and a processor initiates a memory read or write of SM memory, the address is compared to the range specified in the SM Range register. If within range, the PB ignores the transaction, even if it compares to one of the normal memory ranges defined in the other PB configuration registers.
58h	Video Buffer Area Enable. When enabled, processor accesses to the video frame buffer (0000A0000h through 0000BFFFFh) are passed to the PCI bus. When disabled, they are ignored. Default in the KX and the compatibility PB is enabled, and disabled in the aux PB.

Pentium Pro Processor System Architecture

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
<p>General info on PAM registers: PAM registers 0 through 6 define the PCI read/write attributes for 14 memory ranges between 512KB (00008000h) and 1MB-1 (0000FFFFh). Each of the ranges can be designated as disabled, read-only, write-only, or read/write. If enabled for read or write and a processor access of that type is detected within the range, the access is passed to the PCI bus. Conversely, if a PCI access of that type is detected, it is ignored. Each PAM register controls the access rights within two regions. Default for aux PB is all ranges disabled. Default for KX and compatibility PB is all areas (except 512KB-1MB) enabled.</p>	
59h	PAM (Programmable Attribute Map) 0. Controls the regions 00008000h through 00009FFFFh (512KB through 640KB -1), and 0000F000h through 0000FFFFh (the BIOS area).
5Ah	PAM 1. Controls the regions 0000C000h through 0000C3FFFh, and 0000C400h through 0000C7FFFh. This is part of the ISA expansion ROM area.
5Bh	PAM 2. Controls the regions 0000C800h through 0000CBFFFh, and 0000CC00h through 0000CFFFFh. This is part of the ISA expansion ROM area.
5Ch	PAM 3. Controls the regions 0000D000h through 0000D3FFFh, and 0000D400h through 0000D7FFFh. This is part of the ISA expansion ROM area.
5Dh	PAM 4. Controls the regions 0000D800h through 0000DBFFFh, and 0000DC00h through 0000DFFFFh. This is part of the ISA expansion ROM area.
5Eh	PAM 5. Controls the regions 0000E000h through 0000E3FFFh, and 0000E400h through 0000E7FFFh. This is part of the BIOS extension area.
5Fh	PAM 6. Controls the regions 0000E800h through 0000EBFFFh, and 0000EC00h through 0000EFFFFh. This is part of the BIOS extension area.
60h-6F	Reserved.
70h	<p>Error Reporting Command. Controls following features:</p> <ul style="list-style-type: none"> • Enable/disable assertion of SERR# when the PB is acting as the PCI initiator and receives a target abort. • Enable/disable assertion of SERR# when the PB is acting as the PCI initiator of a write transaction and a parity error is detected by the target (it asserts PERR#) when it receives the write data. • Enable/disable assertion of SERR# when PB is acting as PCI initiator of a read transaction and a parity error is detected by the PB (it asserts PERR#) when it receives the read data. • Enable/disable assertion of SERR# when the PB detects PCI address phase parity error. • Enable/disable assertion of PERR# when the PB receives a bad data item on a read, or when being written to by another PCI master.

Chapter 25: 450GX and KX Chipsets

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
71h	Error Reporting Status. Writing a 1 to 1 clears a set status bit. These bits report the following events: <ul style="list-style-type: none"> Parity error detected when writing data to a target. Parity error detected when reading data from a target. When another PCI master initiated a transaction, an address phase parity error detected. Shutdown Special transaction detected on host bus. This bit only used in KX and compatibility PB.
72h-77h	Reserved.
78h-79h	Memory Gap Range. In combination with the <i>Memory Gap Upper Address</i> register, defines a memory address range defined as a "hole" in main memory space. Host accesses within this range are to be directed to the PCI bus. PCI accesses within the hole are not passed to the host bus. The hole may be from 1MB to 32MB in size (in powers of two) and may be enabled or disabled. Default = disabled.
7Ah-7Bh	Memory Gap Upper Address. See <i>Memory Gap Range</i> .
7Ch-7Fh	PCI Frame Buffer. Default = disabled. Use to define the following: <ul style="list-style-type: none"> Frame buffer start address (aligned on 1MB boundary). Enable/disable assignment of frame buffer's attributes (defined in this register) to the VGA video buffer (resides in 0000A0000h through 0000BFFFFh). Enable/disable frame buffer decoder in PB. Enable/disable locked accesses to PCI frame buffer. Flush/do not flush inbound data buffer on non-deferred frame buffer reads. Frame buffer size.
80h-87h	Reserved.
88h-8Bh	High Memory Gap Start Address. In combination with the <i>High Memory Gap End Address</i> register, defines a memory address range defined as a "hole" in main memory space. Host accesses within this range are to be directed to the PCI bus. PCI accesses within the hole are not passed to the host bus. The hole may be any size and may be enabled or disabled. Default = disabled.
8Ch-8Fh	High Memory Gap End Address. See <i>High Memory Gap Start Address</i> .
90h-97h	Reserved.

Pentium Pro Processor System Architecture

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
98h-9Bh	<p>KX - Reserved. The KX PB passes all processor-generated IO transactions to the PCI bus except for accesses to the configuration address and data ports and the TRC register. No PCI-initiated IO transactions are passed to the host bus.</p> <p>GX - IO Space Range #1. In a single PB GX system, the compatibility PB passes all processor-generated IO transactions to the PCI bus except for accesses to the configuration address and data ports and the TRC register. No PCI-initiated IO transactions are passed to the host bus. In a dual-PB GX system, the IO Space Range registers have opposite effects in the two PBs:</p> <ul style="list-style-type: none"> • The compatibility PB passes all processor-initiated IO transactions to the PCI bus <i>except</i> those that fall within either of the two programmed ranges, and those that target the configuration address and data ports and the TRC register. • The aux PB ignores all processor-initiated IO transactions <i>except</i> those that fall within either of the two programmed ranges.
9Ch	PCI Reset. Permits the programmer to assert the PCI RESET signal under program control.
9Dh-9Fh	Reserved.
A0h-A3h	<p>KX - Reserved. See definition of IO Space Range #1.</p> <p>GX - IO Space Range #2. See definition of IO Space Range #1.</p>
A4h-A7h	<p>IO APIC Range. Each PB may have one or more IO APIC modules residing beyond the bridge. The processors use memory accesses to access the registers within the IO APICS. Each PB must therefore be programmed to recognize the range of memory addresses associated with the IO APICs that reside beyond the bridge. This register is used to specify:</p> <ul style="list-style-type: none"> • IO APIC base address (defaults to FEC0000h). • Lowest IO APIC unit number (in the range 0h through Fh). • Highest IO APIC unit number. • Enable/disable this range.
A8h-AFh	Reserved.

Chapter 25: 450GX and KX Chipsets

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
B0h-B1h	<p>Configuration Values Driven on Reset. The Pentium Pro processor (and other host bus agents) sample a number of the lower address lines on the trailing-edge of reset to determine a number of their operational characteristics. When the system is first powered up, reset forces all zeros into this register. The programmer can then write the desired value into this register and then use the TRC register to assert host bus RESET# under program control. The KX or compatibility PB asserts RESET#, drives the contents of this register onto the appropriate host bus signal lines, and then deasserts RESET#. It should be noted that the programmed reset does not clear this register. The <i>Captured System Configuration Values register</i> latches the values from the host bus on the trailing-edge of RESET# and can be read at any time to determine startup configuration options. The operational characteristics that can be programmed via this register are:</p> <ul style="list-style-type: none"> • APIC cluster ID • BINIT# input enable (i.e., BINIT# observation policy). • BERR# input enable (i.e., BERR# observation policy). • AERR# input enable (i.e., AERR# observation policy). • In-Order Queue depth of 1 or 8. • POST entry point of 0FFFFFF0h or 0000FFFF0h. • FRC mode enable or disable.
B2h-B3h	Reserved.
B4h-B5h	Captured System Configuration Values. See description of <i>Configuration Values Driven on Reset</i> register.
B6h-B7h	Reserved.
B8h-BBh	SMM Range. See description of <i>SMM Enable</i> register.
BCh	High BIOS. Allows the programmer to enable/disable PB's ability to pass processor-initiated memory accesses within the 0 through 512KB and the high BIOS area (2MB area from 0FFE0000h through 0FFFFFFFh) to the PCI bus. In reality, the high BIOS area is typically only 128KB in size, but the Pentium Pro processor's internal MTRR register that covers this range has 2MB granularity.
BDh-BFh	Reserved.

Pentium Pro Processor System Architecture

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
C0h-C3h	<p>PB Extended Error Reporting Command. Use this register to:</p> <ul style="list-style-type: none"> • have PB report errors associated with host bus initiated transactions via either a Hard Fail response in the response phase, or by asserting SERR# on the PCI side (typically causes the E/ISA bridge to assert NMI to processor). Defaults to SERR#. • report master aborts, or don't. If programmer set up the PB not report master aborts, the PB returns all Fs for a read that master aborts. For a write, it reports good completion. Defaults to not report master aborts. • enable/disable reporting (via BERR#) of uncorrectable host bus data errors. Default is disable. • enable/disable correcting of single-bit data errors on host bus. Default is disable. Default is disable. • enable/disable reporting of host bus timeouts (due to no target claiming transaction). Default is disable. • enable/disable host bus timeout. If enabled, PB claims host transactions that are not claimed within either 1.5ms or 30ms (programmed via PB Configuration register). Default is disabled. • enable/disable assertion of SERR# on detection of AERR# during error phase of host transaction. Default is disabled. • enable/disable assertion of SERR# on detection of BERR# during error phase of host transaction. Default is disabled. • enable/disable assertion of BINIT# on detection of BERR# during error phase of host transaction. Default is disabled. • enable/disable assertion of BINIT# on detection of host bus protocol error. Default is disabled. • enable/disable assertion of AERR# (during error phase) due to host request phase parity error. Default is disabled on aux PB, enabled on KX and compatibility PB.
C4h-C7h	<p>PB Extended Error Reporting Status. Used by the PB to report the following error conditions:</p> <ul style="list-style-type: none"> • Received Hard Failure response during host bus transaction. • Detected an address parity error when another agent issued a request on the host bus. • Detected an parity error on REQ[4:0]# when another agent issued a request on the host bus. • Detected a correctable data error on the host bus (not implemented in KX PB). • Detected a protocol error on the host bus. • Detected an uncorrectable data error on the host bus (not implemented in KX PB). • Timeout detected on host bus (not implemented in aux PB). • Detected BINIT# asserted on host bus. • Detected AERR# asserted on host bus. <p>The programmer clears a one bit by writing a one to it.</p>

Chapter 25: 450GX and KX Chipsets

Table 25-7: PB Configuration Registers (Continued)

Offset	Description
C8h-CBh	<p>PB Retry Timers. Permits programmer to set up both host bus and PCI bus retry counters.</p> <ul style="list-style-type: none">• Host Retry Counter. 16-bit field to program host retry count in host clocks. A count of 0000h disables it. If a PCI master attempts to read from system memory and is retried by the PB, the PB suspends posting of PCI-to-main memory writes. The PB will reen-able posting of PCI-to-main memory writes until either the retried master successfully retries the transaction or this count expires, whichever comes first.• PCI Retry Counter. If the PB attempts a PCI transaction for a host bus agent and is retried, posting of processor memory writes to PCI is suspended until either the transaction is successfully retried or this counter expires, whichever comes first. Counter can be set up for from 0, 16, 32, 64, or 128 PCI retry attempts.
CCh-FFh	Reserved.

Pentium Pro Processor System Architecture

Figure 25-6: PB Configuration Registers

Byte				Dword
3	2	1	0	Number
Device ID		Vendor ID		0
PCI Status		PCI Command		1
Class Code			Rev. ID	2
BST	Header Type	Latency Timer	Cache Line Size	3
Reserved				4
Reserved				5
Reserved				6
Reserved				7
Reserved				8
Reserved				9
Reserved				10
Reserved				11
Reserved				12
Reserved				13
Reserved				14
Reserved				15
Top of System Memory				16
Reserved				17
SubOrd Bus Num	Bus Number	Device Number	PCI Decode Mode	18
Reserved			PB Configuration	19
CPU R/W Control	Reserved	DeTurbo Counter Control	Reserved	20
SMM Enable	Reserved	PCI R/W Control		21
PAM2	PAM1	PAM0	Video Buffer Area Enable	22
PAM5	PAM3	PAM4	PAM6	23
Reserved				24
Reserved				25
Reserved				26
Reserved				27
Reserved	Error Reporting Status		Error Reporting Command	28
Reserved				29
Memory Gap Upper Address		Memory Gap Range		30
PCI Frame Buffer				31
Reserved				32
Reserved				33
High Memory Gap Start Address				34
High Memory Gap End Address				35
Reserved				36
Reserved				37
IO Space Range #1 in CX, Reserved in IX				38
Reserved			PCI Reset	39
IO Space Range #2 in CX, Reserved in XX				40
IO ATC Range				41
Reserved				42
Reserved				43
Reserved	Configuration Values Driven on Reset			44
Reserved	Captured System Configuration Values			45
SMM Range				46
Reserved			High BIOS	47
PB Extended Error Reporting Command				48
PB Extended Error Reporting Status				49
PB Retry Timers				50
Dwords 51-63 Reserved				63

Chapter 25: 450GX and KX Chipsets

Memory Controller Configuration Registers

Figure 25-6 on page 488 illustrates the PB's configuration registers, while Table 25-8 on page 489 provides a description of each of them.

Table 25-8: MC Configuration Registers

Offset	Description
00h-01h	Vendor ID. Intel vendor ID = 8086h
02h-03h	Device ID. PB device ID = 84C5h.
04h-05h	PCI Command. MC's PCI Command register is hardwired to all zeros.
06h-07h	PCI Status. Hardwired to value 0080h, indicating that the MC supports fast back-to-back transactions.
08h	Revision ID.
09h-0Bh	Class Code. Hardwired to value 050000h, indicating that it's a RAM memory controller.
10h-3Fh	Reserved. Bridge does not implement: Cache Line Size, Master Latency Timer, Header Type, BIST, any BARs, CardBus CIS Pointer, Subsystem Vendor ID, Subsystem ID, Expansion ROM BAR, Interrupt Line, Interrupt Pin, Min_Gnt, or Max_Lat registers.
Remainder of registers are device-specific, outside the scope of the PCI specification.	
40h-43h	MC Base Address Register. Only implemented in the GX MC. The default base address for MC 0 is 000000000h, while that for MC 1 is 100000000h (4GB). Each MC samples its respective OMCNUM pin on the leading-edge of PWRGD to determine whether its MC 0 or 1.
44h-48	Reserved.
49h	Controller Device Number. This read-only register provides the MC's PCI physical device number. Established at powerup when both MCs sample OMCNUM at the leading-edge of PWRGD. <ul style="list-style-type: none"> KX MC, value always 14h (20d). GX MCs, value for MC 0 = 14h (20d), while value for MC 1 = 15h (21d).
4Ah-4Bh	Reserved.

Pentium Pro Processor System Architecture

Table 25-8: MC Configuration Registers (Continued)

Offset	Description
4Ch-4Fh	Command. Controls DRAM configuration, memory controller operations, and reports In-Order Queue depth (captured from A[7]# on trailing-edge of reset. Features controller include: <ul style="list-style-type: none"> • KX MC permits selection of 1- or 2-way interleaved memory architecture. This bit field reports current selection. Default = none. • GX MC permits selection of 1-, 2-, 3-, or 4-way interleaved memory architecture. Default = none. • Hold DRAM page open or close it (default). • Enable/disable of common CAS feature. Default is disabled. • Enable/disable read-around-write feature. Default is disabled. • Enable/disable memory address bit permute feature. Default is disabled. • KX MC permits selection of 1- or 2-way interleaved memory architecture. This bit field used to make selection. Default = 1-way. • GX MC permits selection of 1-, 2-, 3-, or 4-way interleaved memory architecture. This bit field used to make selection. Default = 1-way. • Select number of wait states to be inserted in each beat of multi-beat read. Default = 3 wait states per beat.
50h-56h	Reserved.
57h	SM RAM Enable. Enables/disables the SM memory range specified in the <i>SM Memory Range</i> register. Default is disabled.
58h	Video Buffer Region Enable. Enables/disables video buffer region (0000A0000h through 0000BFFFFh). Default is disabled.
General info on PAM registers: PAM registers 0 through 6 define the MC read/write attributes for the 14 memory ranges between 512KB (000080000h) and 1MB-1 (0000FFFFFh). Each of the ranges can be designated as disabled, read-only, write-only, or read/write. If enabled for read or write and an access of that type is detected within the range, the access is passed to main memory. Each PAM register controls the access rights within two regions. Default for MC is all ranges disabled except for the BIOS area.	
59h	PAM (Programmable Attribute Map) 0. Controls the regions 000080000h through 00009FFFFh (512KB through 640KB -1), and 0000F0000h through 0000FFFFFh (the BIOS area).
5Ah	PAM 1. Controls the regions 0000C0000h through 0000C3FFFh, and 0000C4000h through 0000C7FFFh. This is part of the ISA expansion ROM area.
5Bh	PAM 2. Controls the regions 0000C8000h through 0000CBFFFh, and 0000CC000h through 0000CFFFFh. This is part of the ISA expansion ROM area.
5Ch	PAM 3. Controls the regions 0000D0000h through 0000D3FFFh, and 0000D4000h through 0000D7FFFh. This is part of the ISA expansion ROM area.
5Dh	PAM 4. Controls the regions 0000D8000h through 0000DBFFFh, and 0000DC000h through 0000DFFFFh. This is part of the ISA expansion ROM area.

Chapter 25: 450GX and KX Chipsets

Table 25-8: MC Configuration Registers (Continued)

Offset	Description
5Eh	PAM 5. Controls the regions 0000E0000h through 0000E3FFFh, and 0000E4000h through 0000E7FFFh. This is part of the BIOS extension area.
5Fh	PAM 6. Controls the regions 0000E8000h through 0000EBFFFh, and 0000EC000h through 0000EFFFFh. This is part of the BIOS extension area.
60h-63h	DRAM Row Limits for Rows 0 through 3. Implemented in KX and GX MC. Used to specify the lower and upper addresses for each of the first four RAM rows.
64h-67h	DRAM Row Limits for Rows 4 through 7. Only implemented in GX MC. Used to specify the lower and upper addresses for each of the last four RAM rows.
70h-73h	Reserved.
74h-77h	Single-Bit Correctable Error Address. Contains the memory address that experienced the error. Permits identification of failing SIMM package. Valid only if the corresponding error status bit is set in the <i>Memory Error Status</i> register.
78h-79h	Memory Gap. In combination with the <i>Memory Gap Upper Address</i> register, defines a memory address range defined as a "hole" in main memory space. Host accesses within this range are to be directed to the PCI bus (the PBs must be programmed accordingly). The hole may be from 1MB to 32MB in size (in powers of two) and may be enabled or disabled. Default = disabled. Must reside between the ranges defined by the Low Memory Gap register, and the range defined by <i>High Memory Gap Start Address</i> and <i>High Memory Gap End Address</i> registers.
7Ah-7Bh	Memory Gap Upper Address. See description of the <i>Memory Gap</i> register.
7Ch-7Fh	Low Memory Gap. Defines a memory address range defined as a "hole" in main memory space. Host accesses within this range are to be directed to the PCI bus (the PBs must be programmed accordingly). The hole may be from 1MB to 32MB in size (in powers of two) and may be enabled or disabled. Default = disabled. Must reside below the range defined by the <i>Memory Gap</i> and <i>Memory Gap Upper Address</i> registers, as well as that defined by <i>High Memory Gap Start Address</i> and <i>High Memory Gap End Address</i> registers.
80h-87h	Reserved.
88h-8Bh	High Memory Gap Start Address. Defines a memory address range defined as a "hole" in main memory space. Host accesses within this range are to be directed to the PCI bus (the PBs must be programmed accordingly). The hole may be any size and may be enabled or disabled. Default = disabled. Must reside above the ranges defined by the <i>Low Memory Gap</i> , <i>Memory Gap</i> and <i>Memory Gap Upper Address</i> registers.
8Ch-8Fh	High Memory Gap End Address. See description of <i>High Memory Gap Start Address</i> register.
90h-A3h	Reserved.

Pentium Pro Processor System Architecture

Table 25-8: MC Configuration Registers (Continued)

Offset	Description
A4h-A7h	IO APIC Range. Each PB may have one or more IO APIC modules residing beyond the bridge. The processors use memory accesses to access the registers within the IO APICS. Each PB must therefore be programmed to recognize the range of memory addresses associated with the IO APICs that reside beyond the bridge. The MC must be programmed to ignore accesses within this range. This register is used to specify: <ul style="list-style-type: none"> • IO APIC base address (defaults to FEC00000h). • Lowest IO APIC unit number (in the range 0h through Fh). • Highest IO APIC unit number. • Enable/disable this range.
A8h-ABh	Uncorrectable Error Address. Contains the memory address that experienced the error. Valid only if the corresponding error status bit is set in the <i>Memory Error Status</i> register. Permits identification of failing SDRAM package
ACH-AFh	Memory Timing. Used to program the MC for RAM refresh timing and memory access timing.
B0h-B7h	Reserved.
B8h-BBh	SDRAM RAM Range. Defines the address range associated with SDRAM memory.
BC h	High BIOS Gap Range. Used to enable or disable the MC's ability to recognize addresses within the 2MB high BIOS range (0FFE00000h through 0FFFFFFFh).
BDh-BFh	Reserved.
C0h-C1h	Memory Error Reporting Command. Used to enable/disable correction of single-bit memory errors, as well as enable/disable of correctable and uncorrectable memory errors.
C2h-C3h	Memory Error Status. Indicates whether a correctable or uncorrectable memory error has occurred.
C4h-C5h	System Error Reporting Command. Used to: <ul style="list-style-type: none"> • control the generation of ECC data • control logging of ECC errors • read startup configuration information regarding AERR#, BERR#, and BINIT# observation. • enable/disable MC's ability to drive AERR#, BERR#, and BINIT#.
C6h-C7h	System Error Status. Reports following error conditions: <ul style="list-style-type: none"> • parity error on address latched in request packets 1 or 2 of a transaction's request phase. • parity error on REQ[4:0]# latched in request packets 1 or 2 of a transaction's request phase. • correctable error logged. • uncorrectable error logged. • host bus protocol error
C8h-FFh	Reserved.

Chapter 25: 450GX and KX Chipsets

Figure 25-7: MC Configuration Registers

Byte				Dword Number
3	2	1	0	
Device ID		Vendor ID		0
PCI Status		PCI Command		1
Class Code			Rev. ID	2
Reserved				3
Reserved				4
Reserved				5
Reserved				6
Reserved				7
Reserved				8
Reserved				9
Reserved				10
Reserved				11
Reserved				12
Reserved				13
Reserved				14
Reserved				15
MC Base Address (GX), Reserved (IOQ)				16
Reserved				17
Reserved		Device Number	Reserved	18
MC Command				19
Reserved				20
SMRAM Enable				21
PAM2	PAM1	PAM0	Video Buffer Region Enable	22
PAM6	PAM5	PAM4	PAM0	23
DRAM Row Limit for Row 3	DRAM Row Limit for Row 2	DRAM Row Limit for Row 1	DRAM Row Limit for Row 0	24
DRAM Row Limit for Row 7	DRAM Row Limit for Row 6	DRAM Row Limit for Row 5	DRAM Row Limit for Row 4	25
Reserved				26
First Single-Bit Error Correctible Address				27
Memory Gap Upper Address		Memory Gap		28
Low Memory Gap				29
Reserved				30
Reserved				31
High Memory Gap Start Address				32
High Memory Gap End Address				33
Reserved				34
Reserved				35
Reserved				36
Reserved				37
Reserved				38
IO APIC Range				39
First Uncorrectable Error Address				40
Memory Timing				41
Reserved				42
Reserved				43
SMRAM Range				44
Reserved			High BIOS Range	45
Memory Error Status		Memory Error Reporting Command		46
System Error Status		System Error Reporting Command		47
Dwords 48-63 Reserved				63

Pentium Pro Processor System Architecture

450KX Chipset

Overview

Refer to Figure 25-8 on page 495. The 450KX chipset is a diminished-capability version of the 450GX chipset that supports up to a total of six devices on the processor bus:

- one or two processors.
- one host/PCI bridge.
- one memory controller that support up to 1GB of non-interleaved or 2-way interleaved memory.

Major Features

The 450KX chipset supports the following features:

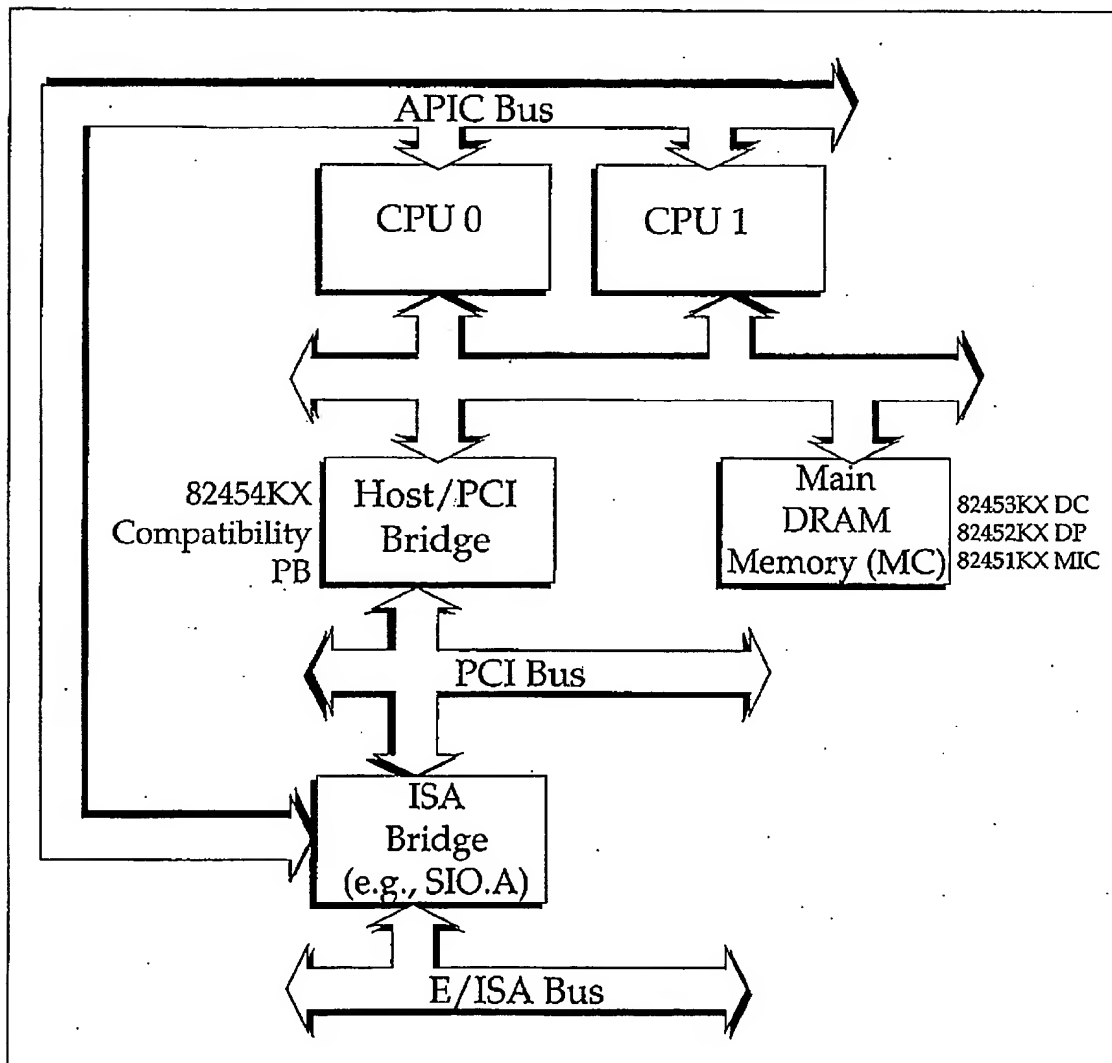
- 60 or 66MHz processor bus speed and 30 or 33MHz PCI bus speed.
- Up to 1GB of 2-way interleaved or 512MB of non-interleaved memory with optional ECC capability.
- Single-bit memory error correction, double-bit or nibble error detection capability.
- Generates parity on the address and REQ[4:0]# signal groups, but not on the data bus.
- Supports 64-bit PCI memory addressing.
- Supports EISA/ISA bridge behind the host/PCI bridge (PB).
- No support for a second PB.
- The memory controller and the PB is highly configurable regarding the memory and/or IO ranges that they recognize when acting as the target of a transaction originated on either side of the PB.
- The memory controller can be programmed to control access to System Management memory.
- Both the PB and the memory controller can be programmed with different accessibility attributes relative to the address ranges that they are configured to recognize.
- When acting as the target of processor-initiated transactions that target PCI devices, the PB can accept up to four transactions in its outbound transaction queue.
- The PB can accept up to four inbound PCI-to-main memory transactions in its inbound transaction queue.
- The memory controller can handle up to four transactions that target main

Chapter 25: 450GX and KX Chipsets

memory.

- The PB contains four, 32-byte outbound data buffers that handle processor-to-PCI memory writes and PCI reads from main memory.
- The PB contains four, 32-byte inbound data buffers that handle PCI-to-main memory writes and processor-initiated reads from PCI memory.
- The memory controller contains four, 32-byte outbound data buffers that handle writes to main memory.
- The memory controller contains four, 32-byte inbound data buffers that handle reads from main memory.

Figure 25-8: Typical Platform Designed Around 450KX Chipset



26

440FX Chipset

The Previous Chapter

The previous chapter provided an overview of the Intel 450GX and 450KX Pentium Pro chipsets.

This Chapter

This chapter provides an overview of the Intel 440FX chipset.

Processor Bus Operation

For a detailed description of the Pentium Pro processor bus operation, refer to:

- "Hardware Section 2: Bus Intro and Arbitration" on page 177
- "Hardware Section 3: The Transaction Phases" on page 239
- "Hardware Section 4: Other Bus Topics" on page 305

PCI Bus Operation

For a detailed description of PCI bus operation, refer to the MindShare book entitled *PCI System Architecture* (published by Addison-Wesley).

ChipSet Overview

Refer to Figure 26-1 on page 504. The Intel 440FX chipset consists of the following components:

- PIIX3 bridge.
- 82441FX PCI and memory controller (PMC).
- 82442FX Data Bus Accelerator (DBX).

Pentium Pro Processor System Architecture

It supports three devices on the processor bus: one or two processors, the PMC, and the DBX. Note that the PMC and DBX together place one device load on the processor bus.

Major Features

The 440FX chipset incorporates the following major features:

- Reduced chip count for low-cost system design.
- Support for one or two processors.
- From 8MB to 1GB of memory.
- Supports 32-bit addressing (not 64-bit addressing).
- Unlike the 450GX and KX chipsets, supports deferred transactions.
- Supports bus frequencies up to 66MHz.
- Supports 5V or 3V DRAMs.
- The PMC incorporates the PCI bus arbiter. The arbitration is hardwired such that it is rotational between the processors, a PCI master (other than the PIIX3), and the PIIX3. Within the PCI masters, it is rotational.
- Complies with the revision 2.1 PCI specification.
- The PIIX3 incorporates the ISA bus bridge, the USB controller, an IDE interface and a link to the IO APIC module.
- When a PCI master accesses main memory, the memory address is supplied to the processors for a snoop.
- The DRAM controller autodetects the DRAM type and size and autoconfigures itself.
- DRAM controller supports ECC or parity (programmable) in memory.
- Supports up to five PCI masters (in addition to the PIIX3).
- The PCI clock is synchronous, divide-by-two with reference to the processor BCLK speed.
- The PMC asserts INIT# to the processor(s) when a shutdown message is received from a processor (via a special transaction). INIT# can also be asserted under program control by writing the appropriate value to the PMC's TRC register.
- BNR# is used by the PMC to throttle the processor(s) from overrunning its IOQ (which can be set a depth of 1 or 4 via strapping option).
- PMC cannot generate the PCI dual-address command (i.e., 64-bit memory addressing isn't supported).
- PMC can be programmed to assert SERR# for: main memory single-bit ECC error; main memory parity error; or main memory multiple-bit ECC error.
- The PMC can be programmed to assert SERR# when it receives a PCI target abort or when PERR# is asserted.
- The DBX asserts BREQ0# during reset to assign agent IDs to the proces-

Chapter 26: 440FX Chipset

sor(s) (for more information, see "Processor's Agent and APIC ID Assignment" on page 42).

- The PMC implements two IO ports: the PCI configuration address and data ports (at IO locations 0CF8h through 0CFBh, and 0CFCh through 0CFFh, respectively).
- Relative to accessing its configuration registers, the PMC is on PCI bus 0, physical device 0, function 0.
- The IDSEL-to-AD line mapping is AD11 asserted for device 0 (this never occurs because the PMC never passes configuration accesses to the PCI bus when it is the targeted device), AD12 for device 1, etc., up to AD31 asserted for device 20.

PMC Configuration Registers

Table 26-1 on page 499 describes the PMC's configuration registers.

Table 26-1: PMC Configuration Register

Offset	Description
00-01h	Vendor ID = 8086h
02-03h	Device ID = 1237h

Pentium Pro Processor System Architecture

Table 26-1: PMC Configuration Register (Continued)

Offset	Description
04-05h	<p>PCI Command register. Bits assigned as follows:</p> <ul style="list-style-type: none"> • Bits [15:10] reserved. • Bit [9] is the fast back-to-back enable bit. Not implemented and hardwired to 0. • Bit [8] is the SERR# enable bit. • Bit [7] is the Stepping control bit. Not implemented and hardwired to 0. • Bit [6] is the PERR# enable bit. • Bit [5] is the VGA Palette Snoop enable bit. It is hardwired to 0. • Bit [4] is the Memory Write and Invalidate enable bit. Not implemented and hardwired to 0. • Bit [3] is the Special Cycle enable bit. Not implemented and hardwired to 0. • Bit [2] is the Bus Master enable bit. This bit is hardwired to 1, permitting the PMC to initiate PCI transactions when necessary. • Bit [1] is the Memory Access enable bit. This bit is hardwired to 1, permitting the PMC to act as the target of PCI memory transactions that target main memory. • Bit [0] is the IO Access enable bit. This bit is hardwired to 0, because the PMC never acts as the target of PCI-initiated IO transactions.
06-07h	<p>PCI Status register. Bits assigned as follows:</p> <ul style="list-style-type: none"> • Bit [15] is the Detected Parity Error bit. • Bit [14] is the Signaled SERR# bit. • Bit [13] is the Received Master Abort bit. • Bit [12] is the Received Target Abort bit. • Bit [11] is the Signaled Target Abort bit. • Bit [10:9] is the DEVSEL# timing field. Hardwired to 01b, indicating that the PMC is a medium decode speed PCI device. • Bit [8] is the Data Parity Reported bit. • Bit [7] is the Fast Back-to-Back Capable bit. Hardwired to 1, indicating that the PMC supports fast back-to-back transactions with different targets. • Bit [6] is the UDF Supported bit. Hardwired to 0, indicating that no diskette is necessary to complete the configuration of the PMC. • Bit [5] is the 66MHz Capable bit. Hardwired to 0, indicating that the PMC cannot operate properly on a PCI bus that runs faster than 33MHz. • Bits [4:0] are reserved and are hardwired to 0.

Chapter 26: 440FX Chipset

Table 26-1: PMC Configuration Register (Continued)

Offset	Description
08h	Revision ID = the rev level (i.e., the stepping) of the PMC silicon.
09-0Bh	Class Code register = 060000h, indicating that the PMC is a host/PCI bridge (class code 06, subclass 00).
0Ch	Cache Line Size register is hardwired to 0 because the PMC "knows" the cache line size (32 bytes).
0Dh	Master Latency Timer register . Must be programmed with the PMC's timeslice when performing a PCI transaction. Lower three bits are hardwired to 0, forcing the programmed value to a value divisible by eight.
0Eh	Header Type register = 00h, bits [6:0] = 0 indicating that the layout of configuration dwords 04d through 15d follows the header type 0 template. Bit [7] = 0 indicating that the PMC is a single-function PCI device.
0Fh	BIST (built-in self-test) register. Hardwired to 00h, indicating that the PMC doesn't implement a BIST.
10-3Fh	Hardwired to 0. The PMC doesn't implement the following registers: Base Address registers, CardBus CIS Pointer, Subsystem Vendor ID, Subsystem ID, Expansion ROM Base Address register, Interrupt Line, Interrupt Pin, Min_Gnt, Max_Lat.
Device-Specific Configuration Registers	
40-4Fh	Hardwired to 0. The PMC doesn't implement device-specific configuration dwords 16d through 19d.
50-51h	PMC Configuration register (PMCCFG) . Allows configuration (or read) of such features as write buffer flushing on interrupts, number of RAS (Row Address Strobe) lines to address DRAM, processor bus frequency, selection of ECC or parity checking for memory accesses, IOQ depth selected.
52h	Deturbo Counter Control (DCC) register . After enabling the Deturbo feature in the TRC register (IO port 93h), this register is programmed with a value that defines how long the PMC keeps BPRI# asserted to keep the processor(s) from using the bus. This features permits the emulation of running the program on a slower machine.

Pentium Pro Processor System Architecture

Table 26-1: PMC Configuration Register (Continued)

Offset	Description
53h	DBX Buffer Control register (DBC). Permits configuration of such features as: <ul style="list-style-type: none"> • retrying a PCI-to-main memory transaction to permit a processor-to-PCI transaction to use the PCI bus. • enable/disable the PMC's ability to post IO writes to the IDE controller (in the PIIX3). • USWC write posting enable/disable. • enable/disable PMC's ability to retry any PCI transaction that takes longer than 32 PCI CLKs. • enable/disable processor-to-PCI memory write posting. • enable/disable pipelining of PCI-to-main memory writes. • enable/disable combining of individual processor-to-PCI memory writes into a PCI burst memory write transaction. • enable/disable ability in the PMC of a main memory read to pass previously-posted main memory writes.
54h	Aux Control register (AXC). Permits selection of: DRAM RAS precharge duration and DRAM address buffer drive strength.
55-56h	DRAM Row Type register (DRT). Used to identify the type of DRAM that populates each row and also identifies empty rows.
57h	DRAM Control register (DRAMC). Select following features: <ul style="list-style-type: none"> • DRAM refresh queuing. • DRAM EDO auto-detect mode enable/disable. • DRAM refresh type: RAS-only, or CAS-before-RAS. • DRAM refresh rate.
58h	DRAM Timing register (DRAMT). Used to select such features as: <ul style="list-style-type: none"> • (for BEDO DRAM) usage of toggle-mode or linear mode data transfer sequence. • select DRAM read data delivery rate at 2-2-2, 3-3-3, or 4-4-4. • select DRAM write data acceptance rate at 2-2-2, 3-3-3, or 4-4-4. • selection of RAS-to-CAS delay.
59-5Fh	Programmable Attribute Map registers (PAM[6:0]). Allows selection of memory R/W attributes within 13 individual ranges within the 640KB-to-1MB range.

Chapter 26: 440FX Chipset

Table 26-1: PMC Configuration Register (Continued)

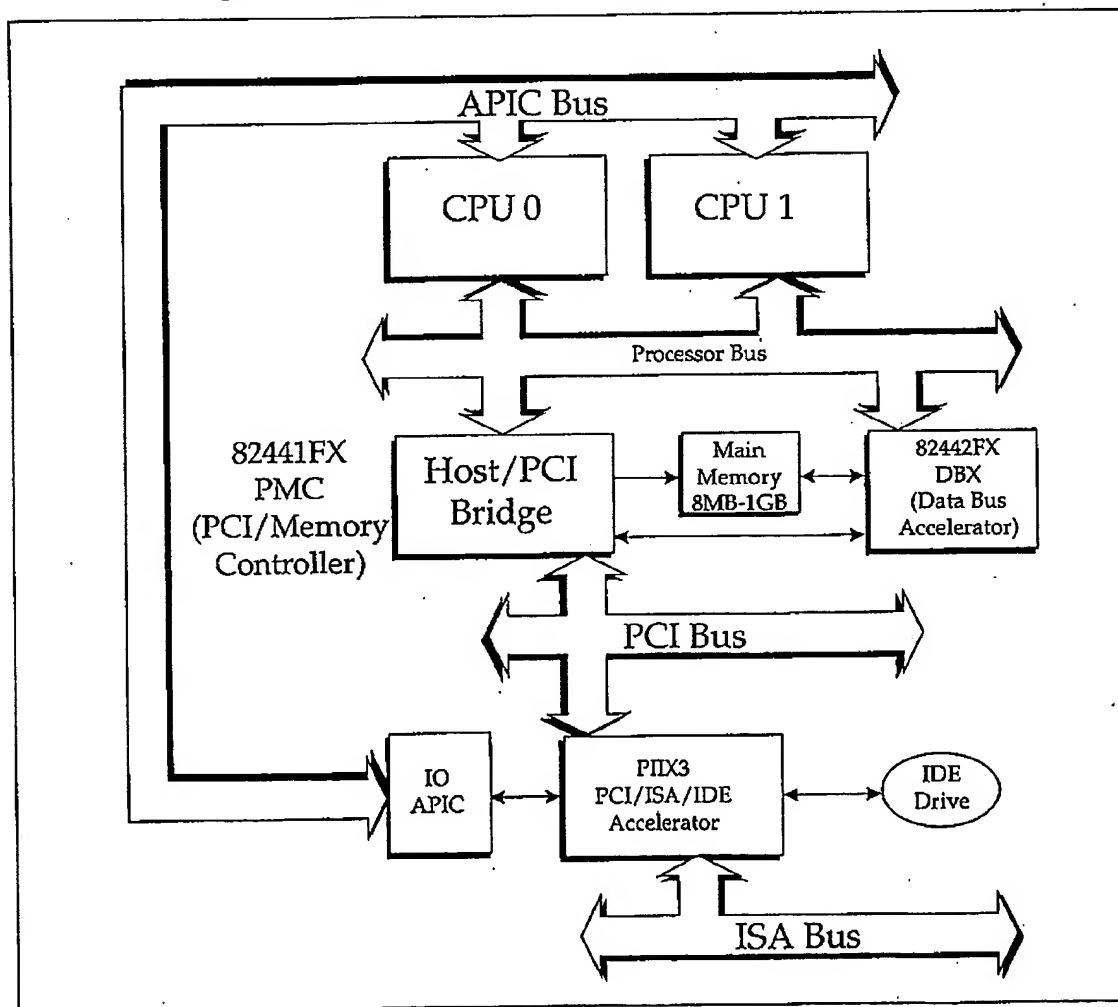
Offset	Description
60-67h	DRAM Row Boundary registers (DRB[7:0]). Used to define the start and end addresses occupied by each row of DRAM modules.
68h	Fixed DRAM Hole Control register (FDHC). Permits the programmer to enable/disable two holes in system memory within the 512KB-640KB and 15-16MB range. When the holes are enabled, processor accesses within these regions are passed to the PCI bus.
69-6Fh	Reserved.
70h	Multi-Transaction Timer register (MTT). Controls the amount of time that a PCI master is permitted to perform back-to-back PCI transactions within a guaranteed time slice programmed into this register.
71h	CPU Latency Timer register (CLT). Permits the PMC to defer a transaction whose snoop phase has been stalled for the amount of time specified in this counter.
72h	SMM RAM Control register (SMRAM). Controls accesses to SMM memory.
73-8Fh	Reserved.
90h	Error Command register (ERRCMD). Controls the PMC's response to various error conditions: <ul style="list-style-type: none"> • enable/disable SERR# assertion on receipt of target abort. • enable/disable SERR# assertion on detection of PERR# asserted. • enable/disable SERR# assertion on multiple-bit or parity error. • enable/disable SERR# assertion on single-bit ECC error.
91h	Error Status register (ERRSTS). Used to report the following error conditions: <ul style="list-style-type: none"> • records row that first multi-bit ECC error occurred in. • records if an uncorrectable ECC error was detected. • records row that first single-bit ECC error occurred in. • records if a correctable single-bit ECC error was detected and corrected.
92h	Reserved.

Pentium Pro Processor System Architecture

Table 26-1: PMC Configuration Register (Continued)

Offset	Description
93h	Turbo Reset Control (TRC) register. Controls the following features: <ul style="list-style-type: none"> enables/disables processor BIST execution at startup. permits programmed hard (RESET#) or soft (INIT#) processor reset. enable/disable of deturbo mode (see "52h" on page 501).
94-FFh	Reserved.

Figure 26-1: Typical Platform Designed Around 440FX Chipset



Appendix

The MTRR Registers

Introduction

The subject of the Memory Type and Range registers (MTRRs) was introduced in the chapter entitled “Rules of Conduct” on page 119. This appendix includes more detail on the MTRRs. All of the MTRRs are implemented as MSRs.

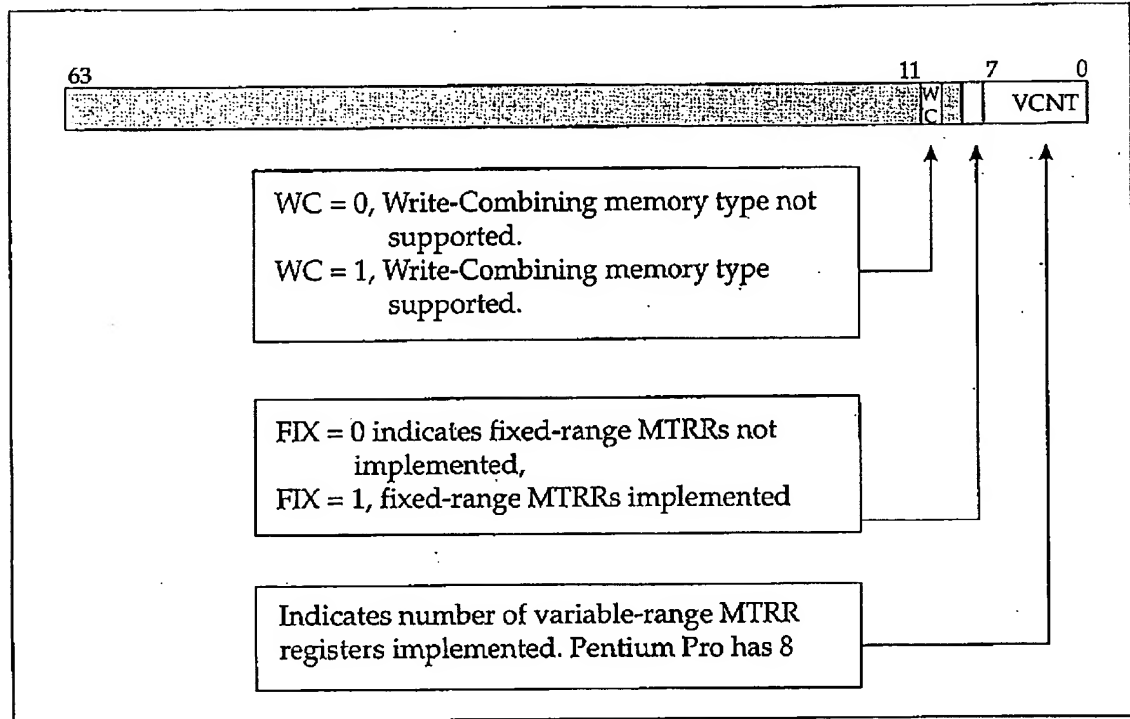
Feature Determination

Use the CPUID instruction with the feature request to determine if the MTRR registers are supported by a processor. EDX[15] = 1 indicates that the MTRRs are supported. Additional information regarding the MTRR registers is obtained by reading the MTRRcap register (MTRR Count and Present register; see Figure 1 on page 506):

- WC = 0 indicates that the WC memory type is not supported, while WC = 1 indicates that it is.
- FIX = 0 indicates that the fixed-range MTRRs aren’t supported, while FIX = 1 indicates that they are.
- VCNT = the number of variable-range MTRRs that are supported.

Pentium Pro Processor System Architecture

Figure A-1: MTRRcap Register

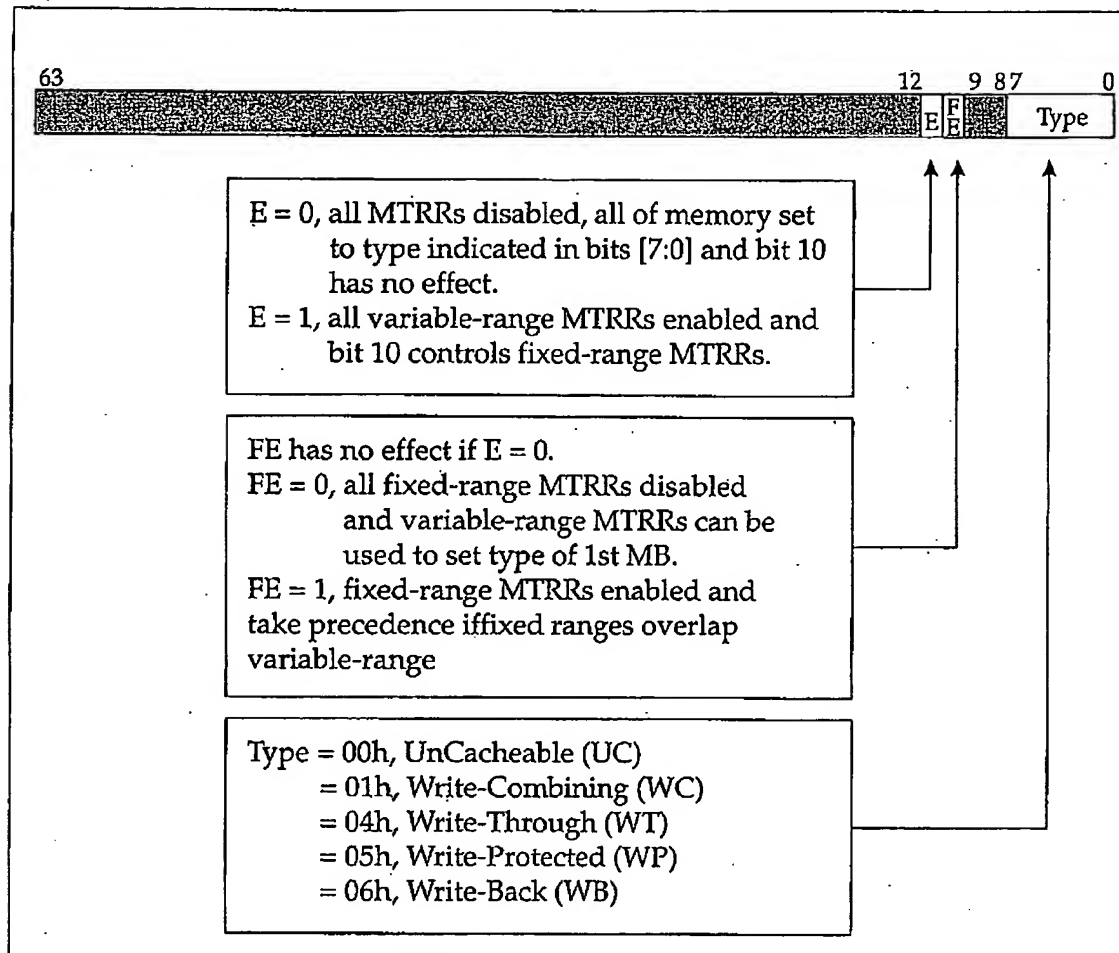


MTRRdefType Register

As described in "Rules of Conduct" on page 119, the MTRRdefType register (see Figure 2 on page 507) defines the memory type for regions of memory not covered by the currently-enabled MTRRs (or for all of memory if the MTRRs are disabled). Reset clears the MTRRdefType register, disabling all MTRRs and defining all of memory as the UC (uncacheable) type.

Appendix: The MTRR Registers

Figure A-2: MTRRdefType Register



Fixed-Range MTRRs

Enabling the Fixed-Range MTRRs

The fixed-range MTRRs are enabled by setting the E and the FE bits = 1 in the MTRRdefType register (see Figure 2 on page 507).

Pentium Pro Processor System Architecture

Define Rules Within 1st MB

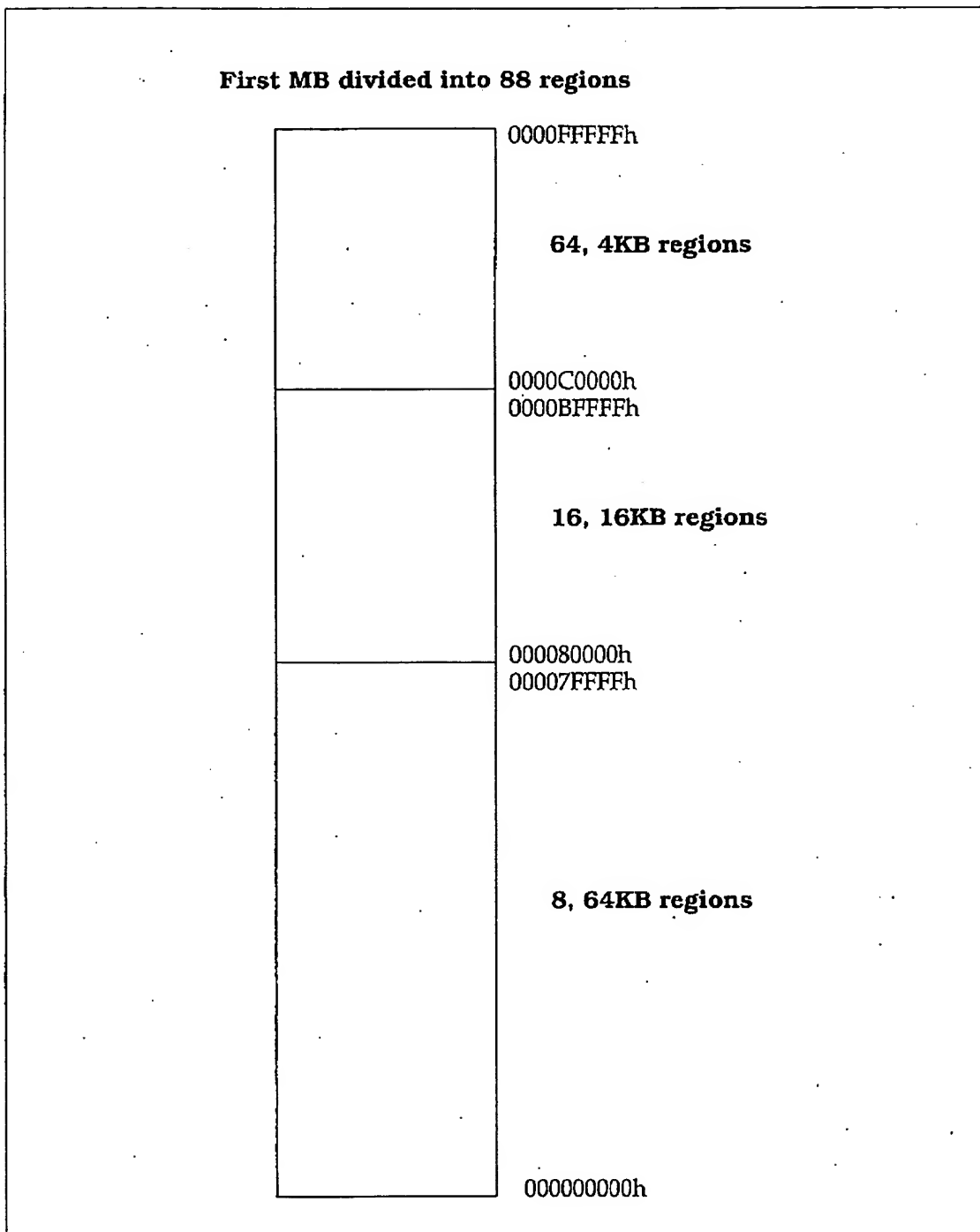
When present and enabled, the fixed-range MTRRs define the rules of conduct within the first MB of memory space (see Figure 3 on page 509). This region is subdivided into 88 subregions by the 11 fixed-range MTRRs (see Table 1 on page 508). Note that each of the fixed-range MTRR registers is 64-bits wide and is subdivided into 8 fields of 8 bits each. The 8-bit value placed in each field defines the memory type (same values as indicated in the TYPE field in Figure 2 on page 507) for the area defined by the bit field.

Table A-1: Fixed-Range MTRRs

63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0	Register
7000-7FFF	6000-6FFF	5000-5FFF	4000-4FFF	3000-3FFF	2000-2FFF	1000-1FFF	0000-0FFF	MTRRfix64K_00000 8, 64KB regions
9C000-9FFFF	98000-9BFFF	94000-97FFF	90000-93FFF	8C000-8FFFF	88000-8BFFF	84000-87FFF	80000-83FFF	MTRRfix16K_80000 8, 16KB regions
BC000-BFFFF	B8000-BBFFF	B4000-B7FFF	B0000-B3FFF	AC000-AFFFF	A8000-ABFFF	A4000-A7FFF	A0000-A3FFF	MTRRfix16K_A0000 8, 16KB regions
C7000-C7FFF	C6000-C6FFF	C5000-C5FFF	C4000-C4FFF	C3000-C3FFF	C2000-C2FFF	C1000-C1FFF	C0000-C0FFF	MTRRfix4K_C0000 8, 4KB regions
CF000-CFFFF	CE000-CEFFF	CD000-CDFFF	CC000-CCFFF	CB000-CBFFF	CA000-CAFFF	C9000-C9FFF	C8000-C8FFF	MTRRfix4K_C8000 8, 4KB regions
D7000-D7FFF	D6000-D6FFF	D5000-D5FFF	D4000-D4FFF	D3000-D3FFF	D2000-D2FFF	D1000-D1FFF	D0000-D0FFF	MTRRfix4K_D0000 8, 4KB regions
DF000-DFFFF	DE000-DEFFF	DD000-DDFFF	DC000-DCFFF	DB000-DBFFF	DA000-DAFFF	D9000-D9FFF	D8000-D8FFF	MTRRfix4K_D8000 8, 4KB regions
E7000-E7FFF	E6000-E6FFF	E5000-E5FFF	E4000-E4FFF	E3000-E3FFF	E2000-E2FFF	E1000-E1FFF	E0000-E0FFF	MTRRfix4K_E0000 8, 4KB regions
EF000-EFFFF	EE000-EEFFF	ED000-EDFFF	EC000-ECFFF	EB000-EBFFF	EA000-EAFFF	E9000-E9FFF	E8000-E8FFF	MTRRfix4K_E8000 8, 4KB regions
F7000-F7FFF	F6000-F6FFF	F5000-F5FFF	F4000-F4FFF	F3000-F3FFF	F2000-F2FFF	F1000-F1FFF	F0000-F0FFF	MTRRfix4K_F0000 8, 4KB regions
FF000-FFFFF	FE000-FEFFF	FD000-FDFFF	FC000-FCFFF	FB000-FBFFF	FA000-FAFFF	F9000-F9FFF	F8000-F8FFF	MTRRfix4K_F8000 8, 4KB regions

Appendix: The MTRR Registers

Figure A-3: First MB of Memory Space



Pentium Pro Processor System Architecture

Variable-Range MTRRs

Enabling the Variable-Range MTRRs

The variable-range MTRRs are enabled by setting the E bit in the MTRRdefType register = 1 (see Figure 2 on page 507).

Number of Variable-Range MTRRs

The number of variable-range MTRRs supported by the processor can be read from the MTRRcap register (see Figure 1 on page 506). The current implementations incorporate eight registers.

Format of Variable-Range MTRR Register Pairs

Each variable-range register actually consists of a register pair—the base and mask registers. The format of a register pair is illustrated in Figure 4 on page 510 and Figure 5 on page 510. Note that the letter *n* indicates the number of the register pair (0-7 for the current processors).

Figure A-4: Format of Variable-Range MTRRphysBasen Register

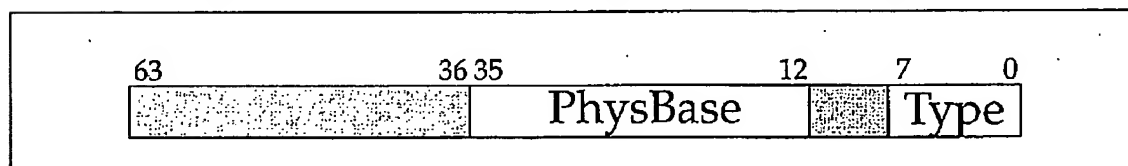
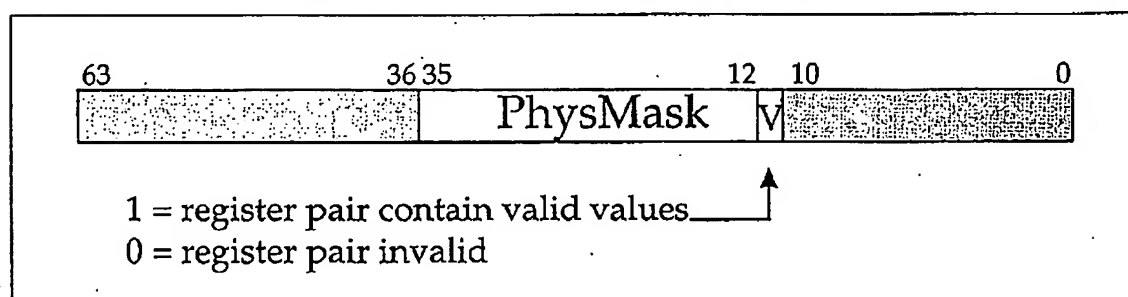


Figure A-5: Format of MTRRphysMaskn Register



Appendix: The MTRR Registers

MTRRphysBase*n* Register

The base register (Figure 4 on page 510) is used to set the base address of the memory region whose memory type is defined by the lower 8 bits of the base register. The minimum granularity of the base address is 4KB-aligned (can't use the lower 12 bits to set the base). The base address assigned must be aligned on an address divisible by the range size defined in the MTRRphysMask register's PhysMask field (see next section).

MTRRphysMask*n* Register

The mask register (Figure 5 on page 510) is used to define the size of the memory range (using the PhysMask field). In addition, the V bit = 0 indicates that the register pair hasn't been set up, while V = 1 indicates that the pair contains valid information.

When 000h is appended to the 6-digit (hex) value in the PhysMask field, the binary-weighted value of the least-significant 1-bit in the mask indicates the size of the memory region defined by the register pair. Note that the PhysMask field must contain all 1-bits from the least-significant 1-bit through bit 35.

Examples

Here are two examples for the variable-range MTRRs:

1. Base = 0000000A20000006h, Mask = 0000000FFF000800h. Indicates that the base address = A20000000h, the memory type = writeback (WB), and the size = 16MB.
2. Base = 0000000549200004h, Mask = 0000000FFFE0800h. Indicates that the base address = 549200000h, the memory type = write-through, and the size is 128KB.

Numerics

16-bit code 117, 118
 2.1 PCI specification 272, 310, 498
 32-bit address memory response agent 249
 32-bit address request agent 249
 32-bit addressing 498
 36-bit address memory response agent 249
 36-bit address request agent 249
 440FX 11, 230, 497
 450GX 11, 211, 230, 461
 450GX/KX chipset 19
 450KX 461, 494
 486 108
 4MB page 381
 4-way system 13
 60x bus 228
 64-bit addressing 498
 64-bit memory addressing 498
 64-bit PCI memory addressing 494
 8259A interrupt controller 334

A

A[12:11]# 41
 A[35:3]# 181, 242, 246
 A10# 40, 48, 346
 A16# 330
 A20M# 41, 348
 A5# 40, 42, 44, 48
 A6# 40, 48
 A7# 40, 48
 A8# 39, 48
 A9# 39, 346
 Aa[35:3]# 243
 Ab[35:3]# 243
 abort class exception 419
 accessed bit 225, 390, 392, 395
 ADC 225
 ADD 225
 address 181
 address parity 246, 250
 address size field 248
 address strobe 202, 207
 address wrap-around 348
 ADDR# 423, 426
 ADS# 202, 207, 214, 219, 233, 242, 246, 251, 289
 advanced programmable interrupt controller 13, 53
 AERR# 39, 46, 195, 253, 289, 310, 314, 323, 345, 434, 435
 AERR# observation 48
 AERR# observation policy 39
 agent 187

agent ID 42, 49, 51, 191, 202, 252, 273, 316, 469, 470, 472, 498
 agent type 251
 aliasing 33, 68
 alignment 116
 allocate-on-write policy 127, 135, 151
 analyzer 340
 AND 225, 449
 AP 56, 58
 AP[1:0]# 242, 246, 253
 AP0# 250
 AP1# 250
 APIC 13, 34, 40, 53, 56, 57, 234, 334, 339, 355, 362, 402, 438, 440, 443, 498
 APIC bus 346
 APIC clock line 351
 APIC cluster ID 48
 APIC data lines 351
 APIC ID 41, 42, 51
 APIC ID number 57
 APIC LVT 373
 APIC module 334
 APIC timer 60
 APICBASE 57
 application processor 56, 58
 arbiter 462, 498
 arbitration 42
 arbitration event 206, 207, 208, 209, 214
 arbitration ID 49, 51, 57
 arbitration logic 202
 arbitration phase 192
 ASZ 247, 248
 ATTR[7:0]# 243, 250
 ATTRIB[7:0]# 127, 129, 251
 Attribute signals 243
 attribute signals 250
 attributes 462
 aux control register 502
 aux PB 20, 467, 469
 AXC 502

B

backside bus 25
 backstop mechanism 113
 bank swapping 172
 BAR 479, 489
 base address register 501
 BCLK 41, 183, 354, 498
 BE[7:0]# 243, 250
 bed-of-nails fixture 41
 BERR# 39, 47, 176, 346, 347, 435
 BERR# assertion/deassertion protocol 347

Index

- BERR# observation policy 39
 - BINIT# 40, 47, 345, 430, 433, 435
 - BINIT# assertion/deassertion protocol 347
 - BINIT# observation policy 40, 48
 - BIOS 59, 120, 121, 406, 427, 473
 - BIOS, high 485
 - BIPI 57, 234
 - BIST 39, 47, 55, 57, 234, 351, 430, 479, 489, 501, 504
 - bit 348
 - bit test and modify instructions 225
 - block next request 231
 - BNR# 231, 498
 - board tester 41
 - boot 13
 - bootstrap inter-processor interrupt 57
 - bootstrap processor 13, 53, 55, 57
 - Boundary Scan interface signals 350
 - BP[3:0]# 349
 - BPM[1:0]# 350
 - BPRI# 20, 211, 313, 323, 462, 470, 471, 481, 501
 - BR[3:1]# 42, 49, 51, 204, 205
 - BR0# 205
 - branch 367
 - branch misprediction 448
 - branch prediction 71, 85, 108
 - branch target address 112, 343
 - branch target buffer 52, 64, 71, 108
 - branch trace message 244, 247, 268, 334, 341, 374, 434, 466, 470, 471
 - branch trap flag 376
 - BRDY# 27
 - breakpoint output pins 349
 - BREQ0# 42, 205, 207, 498
 - BREQ1# 205
 - BREQn# 223
 - bridge device number 480
 - bridge, PCI-to-EISA 334
 - bridge, PCI-to-ISA 334
 - BSP 13, 55, 57, 59
 - BTB 52, 64, 71, 88, 108, 112, 141
 - BTC 225
 - BTR 225
 - BTS 225
 - built-in self-test 39, 234, 430, 479, 501
 - bus analyzer 340
 - Bus Clock 354
 - bus cycle 416
 - bus error 415
 - bus interface 355
 - bus locking 227
 - bus number 480
 - bus request signal 205
 - BUSCHK# 416
 - busy 204, 206
 - busy flag of a TSS 225
 - busy or idle 204
 - byte enable signals 243
 - byte enables 18, 250
- C**
- cache 429
 - cache banks 140
 - cache error 416
 - cache line locking 227
 - cache line size 479, 489, 501
 - cache pipeline stages 155
 - CACHE# 120
 - caches 361
 - call instruction 112
 - captured system configuration values 485
 - CardBus CIS pointer 479, 489, 501
 - castout 142, 153, 166, 171, 381, 398
 - central agent 338, 346
 - central agent transaction 333, 334
 - checker 42, 44, 347
 - chipset 120, 176, 334, 417
 - class code 479, 489, 501
 - CLI 335, 404, 409, 413
 - clock 355, 438
 - Clock frequency ratio 49
 - clock, APIC 351
 - CLT 503
 - cluster 13, 41, 42, 48, 55, 202, 270, 274, 462
 - cluster bridge 274, 462
 - cluster ID 41
 - CMOV 362, 367
 - CMOV instruction 115
 - CMPXCG8B 225
 - CMPXCHG 225
 - code cache 52, 133, 135, 264
 - code cache lookup 140
 - code segment register 53
 - code TLB 140
 - command 478, 489, 490, 500
 - Compare and Exchange 8 Bytes instruction 362
 - compatibility bridge 337, 346
 - compatibility PB 20, 466, 469
 - complex decoder 66
 - conditional move instruction 367
 - configuration address and data ports 499
 - configuration address port 466, 470, 471, 472, 473, 476
 - configuration data port 466, 476
 - configuration mechanism 476

configuration register 501
 configuration registers 468, 476, 478
 configuration transaction 18
 configuration value register 473
 configuration values driven on reset 485
 configure 37
 CONFVR 473
 CONFVR register 471
 context switch 117
 correctable ECC error 435
 corrected error 425
 counter mask 440
 CPU ID 49
 CPU latency timer 503
 CPU Present 352
 CPU R/W Control 481
 CPUID 116, 125, 137, 359, 366, 369, 371, 417, 428, 438, 451, 507
 CPUPRES# 352
 CR0 52, 64, 90, 149
 CR0[CD] 55, 59, 90
 CR0[NW] 55, 59, 64, 90
 CR0[TS] 453
 CR0[WP] 390, 392
 CR2 53
 CR3 53, 377, 384, 387, 392, 395, 398
 CR4 52, 376
 CR4[MCE] 256, 416, 419
 CR4[PCE] 368, 442
 CR4[PGE] 398
 CR4[PSE] 381
 CR4[PVI] 407, 413
 CR4[TSD] 370
 CR4[VME] 404
 critical quadword 155, 164, 165, 166, 167, 171, 280
 CS 53
 CXS 362

D

D[63:0]# 181, 282
 data bus 181
 data bus ECC/parity protection bits 282
 data bus error checking 46
 data cache 52, 133, 134, 143
 data lines, APIC 351
 data path 165
 data path unit 462, 469
 data phase 192, 196, 197, 281
 data segment register 53, 117
 data TLB 146
 data transfer 196
 DBC 502

DBSY# 270, 272, 281, 309, 310
 DBX 497, 498
 DBX buffer control 502
 DC 462, 469
 DCC 471, 501
 DE 376
 debug breakpoint address registers 54
 debug control register 54, 341
 debug extension 362, 376
 debug status register 54
 debug tool 231, 232, 341, 343, 466
 DEBUGCTL 341
 DebugCTL 374, 375
 DebugCTL[BTF] 376
 DEC 225
 DEC1 69, 84
 DEC2 69, 85, 113
 decoder 66
 decoder zero 66
 decoders one and two 66
 defer 278, 503
 defer enable 253, 268, 316
 DEFER# 260, 268, 269, 272, 274, 317, 475
 deferral 307
 deferred 260, 280, 282
 Deferred ID 243
 deferred ID 250, 251, 313, 321, 323
 deferred interrupt data structure 405
 deferred read 311
 deferred reply 244, 247, 268, 309, 311, 316
 deferred reply transaction 191, 273, 281
 deferred response 196, 260, 311
 deferred transaction 189, 273, 317, 498
 deferred transaction queue 317, 318, 346
 deferred write 320
 delivery mode 402
 delivery status 402
 DEN# 253, 268, 316, 317, 321, 337, 343
 DEP[7:0]# 282
 descriptor 364
 deturbo counter 481
 deturbo counter control 471, 501
 device configuration 476
 device ID 478, 489, 499
 device not available exception 453
 DEVSEL# 272, 310, 317, 319, 323
 DID 250, 273, 317, 321, 325, 475
 DID[7:0]# 243, 250, 251
 differential receiver 180
 dirty bit 225, 392, 395
 DIS 70

Index

DMA channel 20
DOS handler 412
DOS task 403
double-bit ECC detection 494
double-fault exception handler 339
DP 93, 462, 469
DR0 54
DR3 54
DR6 54
DR7 54
DRAM architecture 472, 473
DRAM control 502
DRAM controller 462, 469, 498
DRAM row boundary registers 503
DRAM row limits 491
DRAM row type 502
DRAM timing 473, 502
DRAMC 502
DRAMT 502
DRB[7:0] 503
DRDY# 270, 272, 282, 309, 310, 343
DRT 502
DS 53
DSZ 247
DTLB 53, 146
dual-address command 19, 498
dual-processor 363
dynamic branch prediction 71, 108, 109, 113, 141

E

E bit 509
E state 14
EAX 54
EBL 433
EBL_CR_POWERON 45, 176, 340, 346, 427
EBP 54
EBX 54
ECC 135, 136, 137, 176, 282, 434, 435, 498, 503
ECC checking 462
ECC error 415
ECC syndrome 435
ECX 54
edge detect bit 440
edge rates 182
EDI 54
EDX 53, 55
EE[63:0] 423
effective impedance 182
EFLAGS 53, 367
EFLAGS[ID] 359
EFLAGS[IF] 405, 413
EFLAGS[IOPL] 406, 409

EFLAGS[NT] 413
EFLAGS[TF] 376
EFLAGS[VIF] 407, 409, 413
EFLAGS[VIP] 407, 410
EFLAGS[VM] 403
EFLAGS[VME] 409
EIP 53
EIPV 422
EISA 19, 211, 270, 309
EISA master 20
electrical high 180
electrical low 180
EMMS 453, 456
Empty MMX State instruction 453
ERRCMD 503
error code, compound 429, 430
error code, compound MCA 429
error code, simple 429
error command 503
error correcting code 135
error handler, FP 348
error instruction pointer valid 422
error logging register banks 416
error phase 192, 195
error reporting command 482
error reporting status 483
error status 503
error, address parity 434
error, bus-related 433
error, external bus-related 433
errors, bus and interconnect 431
errors, memory hierarchy 431
ERRSTS 503
ES 53
ESI 54
ESP 54
event counter 402
event select 441
event select registers 54
event type, duration of an event 439
event type, occurrences of 439
eviction 432
EX 70, 88, 93
exception 18d 416
exceptions 451
exchange instructions 225
exclusive state 144
EXF[4:0]# 243, 250
EXF0# 253
EXF1# 253
EXF2# 253

EXF3# 253
 EXF4# 252
 expansion ROM BAR 479, 489, 501
 extended error reporting command 486
 extended error reporting status 486
 extended function signals 243
 extended functions 250
 extended request 250
 external bus 429
 external bus unit 166
 external interrupt 53
 ExtINT 402

F

fault 339
 FCMOV 367
 FCMOV instruction 115
 FCOMI 367
 FCOMIP 367
 FDHC 503
 FE bit 509
 features 361
 feed forwarding 33, 68, 104
 FERR# output 348
 final boot inter-processor interrupt 58
 FIPI 58
 FIX 507
 fixed DRAM hole control 503
 floating-point 348
 floating-point conditional move 367
 flush 125, 252, 339, 466
 flush acknowledge 252, 339, 466
 Flush Acknowledge special transaction 354
 FLUSH# 41, 47, 339, 354
 FP error handler 348
 FPU error 348
 FPU present 362
 FPU registers 451, 453
 frame buffer 380
 FRAME# 272, 310
 FRC 40
 FRC error 430
 FRC mode 42, 48
 FRCERR 44, 347, 434
 free 231, 233
 FRSTOR 453
 FS 53
 FSAVE 453
 FUCOMI 367
 functional redundant checker 40
 FWAIT instruction 348

G

GAT 20
 GDTR 54
 global bit 392, 395
 global descriptor table register 54
 global page 379, 395, 401
 global page feature 362, 398
 global registers 419, 421
 GOTO 116
 GP exception 371, 404, 406, 410, 412, 413, 442
 Ground 353
 GS 53
 GTL 179
 GTL+ 179, 353
 guaranteed access time 20
 Gunning transceiver logic 179

H

HAL 121
 halt 252, 339, 438, 467, 471
 handler 409
 hard failure 196, 278, 280, 282, 291, 435
 hard failure response 273
 Hard reset 350
 hardware interrupt 409, 410
 header type 479, 489, 501
 high 180
 high BIOS gap range 492
 high memory gap end address 483, 491
 high memory gap start address 483
 hit on a modified line 260
 hit on an E or S line 260
 HIT# 260
 HITM# 210, 260
 hold time 183
 host retry counter 487
 host/PCI bridge 17, 188, 211, 270, 309, 311
 host/PCI bridges 462

I

I state 14
 IA 29
 IA instructions 29, 67
 IA register set 31
 Icache 137
 ID queue 85
 IDE interface 498
 idle 204, 206, 208, 256, 279, 303
 idle ownership state 346
 IDSEL 499
 IDT 402, 419, 443
 IDTR 54

Index

- IBRR# 176, 347, 435
 - IF bit 335, 404
 - IFU 434
 - IFU1 64, 69, 71, 85, 109
 - IFU2 71, 108, 112, 113, 138
 - IFU3 69, 138
 - IGNNE# 41, 348, 349
 - impedance 182
 - implicit writeback 167, 196, 276, 278, 280, 292
 - IN 331
 - INC 225
 - indirect branches 116
 - INIT# 39, 47, 347, 351, 498, 504
 - initiator 187, 231
 - in-order queue 40, 199, 200, 230
 - INS 331
 - instruction boundaries 71
 - instruction decode queue 52, 55
 - instruction fetch unit 434
 - instruction pointer 53
 - instruction pool 52, 55
 - INT 7 453
 - INT n 404, 406
 - In-Target Probe 350
 - intel architecture 29
 - interconnect 429
 - inter-device spacing 182
 - interleaved 172
 - internal error 347
 - inter-processor interrupt 14
 - interrupt 409
 - interrupt acknowledge 244, 247, 268, 334, 402, 434, 467
 - interrupt acknowledge transaction 18
 - interrupt controller 18, 334, 348, 409
 - interrupt descriptor table 402, 419, 443
 - interrupt descriptor table registers 54
 - interrupt enable bit 440
 - interrupt line 479, 489, 501
 - interrupt pin 479, 489, 501
 - interrupt redirection bitmap 410, 412
 - interrupt request 334
 - interrupt table 413
 - interrupt vector 18, 409
 - interrupts 451
 - INTR 334, 349, 351
 - invalid state 144
 - INVD 117, 339
 - invert bit 440
 - INVLPG 117
 - IO access 433
 - IO address range 330
 - IO APIC 34, 334
 - IO APIC module 334, 498
 - IO APIC range 484, 492
 - IO decoders 471
 - IO instruction 125
 - IO port F0h 349
 - IO read 244, 247, 268, 434
 - IO space range #1 484
 - IO space range #2 484
 - IO space range registers 471
 - IO transactions 329
 - IO write 244, 247, 268, 434
 - IOGNT 211
 - IOGNT# 20, 462, 469, 471
 - IOQ 40, 48, 199, 200, 230, 255, 270, 272, 345, 498
 - IOREQ# 20, 211, 462, 469, 471
 - IPI 339
 - IRET 117, 125, 405, 419
 - IRQ13 handler 348
 - ISA 19, 211, 270, 272, 309, 310, 348, 462
 - ISA device 467
 - ISA IO aliasing 479
 - ISA master 20
 - ITLB 53
 - ITP 350
- I**
- JEU 85, 88, 109, 112
 - jump execution unit 85, 109, 112
- K**
- KEN# 120
 - kernel 13, 60, 380
 - kill 228
 - Klamath 135
 - KX 484
- L**
- L1 155
 - L1 caches 133
 - L1 code cache 52, 364
 - L1 data cache 52, 143, 364
 - L2 155
 - L2 cache 34, 52, 133, 135, 141, 162, 365, 434
 - L2's relationship with the L1 code cache 148
 - L3 155
 - L3 cache 129
 - L4 155, 167
 - L5 155, 167
 - L6 155, 167
 - LastBranch 374
 - LastBranchFromIP 375

- LastBranchToIP 375
 - LastException 374
 - LastExceptionFromIP 375
 - LastExceptionToIP 375
 - latency timer 479
 - LBR 375
 - LDTR 54
 - least-recently used 142, 153, 399
 - legacy architecture 29
 - LEN 247, 287
 - LGDT 117
 - LIDT 117
 - line 14
 - linear address 140, 174
 - linear, or logical, address 383
 - LINT0 41, 334
 - LINT0/INTR 351
 - LINT1 41
 - LINT1/NMI 351
 - LLDT 117
 - load 104, 118, 155, 159
 - load pipeline 104, 106
 - local APIC 34, 234
 - local descriptor table register 54
 - Local Interrupt input 0 351
 - Local Interrupt input 1 351
 - local vector table 402, 443
 - lock 223, 434
 - lock prefix 225
 - LOCK# 213, 223, 225, 227, 274
 - locked operation 125
 - lookup, code cache 140
 - low 180
 - low memory gap 491
 - low power enable 49
 - Low Power Enable bit 340
 - low-voltage swing 179
 - LRU 142, 152, 166, 171, 399
 - LTR 117
 - LVS 179
 - LVT 373, 402, 443
- M**
- M state 14
 - machine check address register 416
 - machine check architecture 54, 362, 374, 415
 - machine check architecture register banks 176
 - machine check architecture registers 346, 347, 419, 428
 - machine check architecture-defined error code 426
 - machine check exception 176, 256, 346, 347, 362, 376, 415, 416, 419
 - machine check global control register 421, 422
 - machine check global count and present register 421
 - machine check global status register 421
 - machine check in progress 422
 - machine check type register 416
 - mantissa 452
 - mask 402
 - Maskable External Interrupt 351
 - mass storage 384
 - master 42, 44, 347
 - master abort 273, 313, 317, 319, 323, 327
 - master latency timer 272, 309, 489, 501
 - master/checker pair 44
 - matrix 446
 - max_lat 479, 489, 501
 - MC 469
 - MC base address register 489
 - MC0_ADDR 427
 - MC0_CTL 427
 - MC0_MISC 427
 - MC0_STATUS 427
 - MC1_ADDR 427
 - MC1_CTL 427
 - MC1_MISC 427
 - MC1_STATUS 427
 - MC2_ADDR 427
 - MC2_CTL 427
 - MC2_MISC 427
 - MC2_STATUS 427
 - MC3_ADDR 428
 - MC3_CTL 428
 - MC3_MISC 428
 - MC3_STATUS 428
 - MC4_ADDR 428
 - MC4_CTL 428
 - MC4_MISC 428
 - MC4_STATUS 428
 - MCA 54
 - MCA error code 426
 - MCA register init 428
 - MCAR 416
 - MCE 376, 416, 419
 - MCG_CAP 421, 427
 - MCG_CTL 421, 422, 427
 - MCG_CTL_P bit 422
 - MCG_STATUS 421, 427
 - MCI_ADDR 423, 426
 - MCI_CTL 423
 - MCI_MISC 423, 426
 - MCI_STATUS 423, 424
 - MCI_STATUS[ADDRV] 423

Index

MCi_STATUS[MISCv] 423
MCIP 422
MCTR 416
memory access 433
memory addressing 494
memory code read 245, 247
memory controller 462, 469, 472, 497
memory controller device number 489
memory data read 245, 247
memory error reporting command 492
memory error status 492
memory gap 491
memory gap range 483
memory gap start address 491
memory gap upper 491
memory gap upper address 483
memory interface components 462, 469
memory read and invalidate 245, 247, 266, 267, 281, 282
memory semaphore 222
memory timing 492
memory type 123
memory type and range registers 14, 54, 119, 121, 507
memory write (may be retried) 245, 247
memory write (may not be retried) 245, 247
memory, interleaved 462
memory, SMM 469
memory, system management 462
memory-mapped IO devices 120
MESI 134, 135, 138, 148, 264
MESI cache protocol 14
message 338
messages 252
micro-op 30, 67, 174, 430
MICs 462, 469
min_gnt 479, 489, 501
misaligned address boundary 226
MISCv 423, 426
mispredicted branch 109, 367
mispredicted branches 115
miss 260
mixed code and data 176
MLT 272, 310
MM[7:0] 451
MMX 445
MMX ALU Unit 456
MMX instruction set 453
MMX instruction syntax 454
MMX Multiplier Unit 456
MMX registers 446, 451
MMX Shifter Unit 456

MMX, web site 456
model specific error code 426
model-specific register 371
model-specific registers 121
modified line 258
modified state 144
MOV 455
MS-DOS environment 348
MSR 45, 54, 371, 416, 419, 426, 442
MTRR 14, 54, 59, 60, 64, 119, 121, 149, 250, 251, 311, 362, 374, 507
MTRR count and present register 507
MTRR, fixed-range 121
MTRRcap 124, 507
MTRRdefType 59, 123, 129
MTRRdefType register 508
MTRRphysBasen Register 513
MTRRphysMaskn Register 513
MTRRs, fixed-range 507, 509, 510
MTRRs, variable range 121, 512
MTRRs, variable-range 512
MTT 503
multi-processing spec 59
multitasking 403
multi-transaction timer 503

N

NA# 27
CR0 348
NEG 225
Next IP 64
nibble error detection 494
NMI 176, 346, 347, 351, 402
no data 278, 280, 310
no data response 273, 343
noise margins 179, 182
non-blocking 135, 136, 162
Non-Maskable Interrupt 351
nop 339
normal data 278, 281, 284, 310
normal data response 273
NOT 225

O

OMCNUM 472, 489
OOO 157
open-drain 181, 231
open-drain signal 253, 260
OR 225, 449
Orion 472
OS 120, 441
other information field 426

OUT 331
 out-of-order 157, 366
 out-of-order execution 30
 OUTS 331
 overdrive processor 363
 overflow 425, 448
 overshoot 182
 ownership state 256

P

packed array 446
 packed data 447
 packed word compare 449
 packet B 268
 packets A and B 243
 PACKSS 455
 PACKUSWB 455
 padd 455
 PADDs 455
 PADDUS 455
 PADDUSW 454
 PAE 362, 376, 377, 379, 381, 383, 385, 394, 401
 page 381, 384
 page cache disable 388
 page directory 225, 380, 381, 384, 398
 page directory base address register 53, 377
 page directory entry 390
 page directory pointer table 385, 387
 page directory pointer table (PDPT) base address register 377
 page fault address register 53
 page global enable 377
 page group 384
 page present 390
 page present bit 392, 394
 page size bit 394
 page size extension 362, 376, 379, 401
 page table 250, 380, 381, 384, 398
 page table entry 120, 225, 251, 276, 392, 398
 page write-through 388
 page, 2MB 394
 pages 140, 146, 164
 pages, global 398
 paging 129, 379, 381, 383
 PAM registers 482, 490, 502
 pand 449, 455
 pandn 449, 455
 parity 195, 253, 282, 494, 498, 503
 parity checking 462
 parity enable 417
 parity error 273, 313, 318, 320, 323, 327, 415, 416
 park 210

parking 204, 221
 PB arbiter selection 480
 PB configuration 480
 PB outputs 374
 PB, aux 467
 PB, compatibility 466
 PB's configuration registers 478
 PBn output pin 440
 PBs 211, 462
 PC Compatibility signal group 348
 PCC 426
 PCD 120, 130, 388, 390, 392, 394
 PCE 377, 442
 PCI 270, 273, 309, 313, 317, 323, 337, 461, 470, 497
 PCI bus arbiter 498
 PCI bus number 480
 PCI clock 498
 PCI command 478
 PCI configuration device number 470
 PCI decode mode 479
 PCI device configuration mechanism 476
 PCI Device Number 472
 PCI frame buffer 483
 PCI master 211
 PCI R/W control 481
 PCI reset 484
 PCI retry counter 487
 PCI specification 272, 310, 498
 PCI status 479
 PCI subordinate bus number 480
 PCI/ISA bridge 337
 PCINT 443
 PCI-to-EISA 466, 467
 PCI-to-ISA 466, 467
 PCLK 354
 PCMPEQ 456
 pcmpeqw 449
 PCMPGT 456
 PDPT 377, 385, 387
 PDPT entry 388
 PEN# 416, 417
 Pentium 27, 108, 120, 138, 189, 192, 361, 368, 370, 371, 373, 381, 403, 415, 416, 437, 439
 Pentium, dual-processor 27
 PerfCtr registers 441
 PerfCtr0 439
 PerfCtr1 439
 PerfEvtSel registers 440, 442
 PerfEvtSel0 439
 PerfEvtSel1 439
 performance counter enable 377

Index

- performance counter interrupt 443
 - performance counter overflow interrupt 402, 443
 - performance counters 441
 - performance counters, starting 443
 - performance counters, stopping 443
 - performance monitoring 34, 368, 373, 374, 402, 437, 439
 - performance monitoring counter 369
 - performance monitoring counters 439
 - performance monitoring or breakpoint 374
 - PERR# 498, 503
 - PGE 377, 379, 401
 - phase-locked loop 41
 - Phase-Locked Loop decoupling pins 354
 - physical address extension 362, 376, 377, 379, 401
 - physical memory address 140
 - PhysMask field 513
 - PIC timer 435
 - PICCLK 351
 - PICD[1:0] 351
 - PICD1 57
 - PIX3 498
 - pin control bit 440
 - pipeline stages 155
 - pipelined 135, 136
 - pipelining 27
 - pixel bytes 446
 - PLL 41, 354
 - PLL[2:1] 354
 - PMADDWD 455
 - PMC 497, 498
 - PMCCFG 501
 - PMULHW 455
 - PMULLW 455
 - point-to-point transaction 333
 - pop 452
 - por 449, 455
 - port F0h 349
 - POST 13, 40, 55, 58, 59, 85, 123, 339
 - posted write 107, 124, 126, 151, 226
 - POWERGOOD 192, 353
 - Power-on restart address 48
 - power-on self-test 13, 85
 - PowerPC 228
 - PRDY# 350
 - prefetch 432
 - prefetch streaming buffer 52, 55, 64, 85, 141, 174
 - prefetcher 64, 134, 174
 - prefix 225
 - PREQ# 350
 - priority 202
 - priority agent 211, 236
 - priority request agent 191
 - priority scheme 191
 - Probe Ready 350
 - Probe Request 350
 - processor context corrupt 426
 - processor type 363
 - processor version information 361
 - programmable attribute map 482
 - programmable attribute map registers 502
 - protected mode 60
 - protected mode handler 409, 412
 - protected mode task 413
 - protected mode virtual interrupt 376
 - protected mode virtual interrupts 407
 - PS 381
 - PS bit 394
 - PSE 362, 376, 379, 380, 381, 394, 401
 - PSLL 455
 - PSRA 455
 - PSRL 455
 - PSUB 455
 - PSUBS 455
 - PSUBUS 455
 - pullup resistor value 182, 183
 - PUNPCKH 455
 - PUNPCKL 455
 - push 452
 - PVI 376, 407, 413
 - PWRGD 469, 472, 489
 - PWT 120, 130, 388, 390, 392, 394
 - PXOR 455
- Q**
- quad 13
- R**
- R/W 392
 - R/W bit 390, 394
 - RAT 70, 86
 - raw code 138
 - RDMSR 121, 123, 362, 371, 439, 442
 - RDPMC 368, 370, 442
 - RDTSC 438
 - read and invalidate 16, 127, 228, 434
 - read parity error 416
 - read performance counter 442
 - read TSC 438
 - read with intent to modify 228, 434
 - read/modify/write 223
 - read/write bit 390, 392, 394
 - ready for retirement 30, 89, 106, 157

real mode 351
 real mode interrupt table 413
 real mode OS 406
 redirection bitmap 410, 412
 Reference voltage 353
 reference voltage 180, 183
 register aliasing 33, 68
 relaxed DBSY# deassertion 287
 REQ# 272, 309
 REQ[4:0]# 241, 242, 246, 251, 462
 REQa[4:0] 243
 REQb[4:0] 243
 request agent 187, 194, 211, 231, 257
 request for ownership 434
 request packet B 249, 252
 request packets A and B 253
 request parity 246, 251
 request phase 192, 195, 201, 242
 request signal group 193, 195, 242, 245, 471
 request type 361
 reservation station 87, 456
 reset 37, 54, 55, 57, 58, 192, 438, 473
 reset effects 52
 RESET# 52, 347, 350, 353, 474, 504
 response 196
 response agent 188, 195, 242, 258
 response bus 278
 response bus parity bit 278
 response bus parity checking 46
 response error 435
 response phase 192, 195, 197, 277
 response phase stall 303
 restart instruction pointer valid 421
 resume 340
 RET1 70, 89
 RET2 71, 90
 retirement 30, 89, 106
 retirement unit 68
 retried 260
 retry 28, 189, 196, 268, 272, 278, 279, 282, 310, 320, 327, 475
 retry timers 487
 return instruction 113
 revision ID 479, 489, 501
 ring 182
 ringing 179
 RIPv 421
 RISC 30, 67
 RMW 223, 225
 ROB 70, 86, 87, 93, 109, 366
 ROB timeout 435

ROB timeout counter 435
 role reversal 317
 ROM 55, 58, 462
 ROM code 126
 ROM parity error, microcode 430
 ROM, boot 58, 462, 470, 471
 rotating ID 204, 206, 256, 346
 rotational priority 202
 RP# 242, 246, 253
 RR 93, 106
 RS 87, 456
 RS[2:0]# 278, 303
 RSB 113
 RSM 117, 125, 340
 RSP# 278, 303
 RT 93
 Rt 182
 rules of conduct 119, 250
 RWITM 228

S

S state 14
 saturated math 448
 SBB 225
 SD 93
 segment descriptor 225, 276
 segment register 117
 self-modifying code 116, 126, 173
 self-snooping 175, 268
 semaphore 222, 276
 serializing 116, 125, 225, 226, 438
 serializing instruction 366, 369, 370
 SERR# 498, 503
 setup time 183
 shadow RAM 126
 shared memory buffer 221
 shared resource 222
 shared state 144
 shutdown 252, 339, 419, 422, 467, 471, 498
 signal groups 193
 SIMD 451
 single-bit correctable error address 491
 single-bit memory error correction 494
 single-quadword, 0-wait state data transfer 301
 single-step debug exception 376
 single-step exception on branches 374
 SIO.A 334
 SIPI 58, 60
 SM RAM enable 490
 SM RAM range 492
 SMI acknowledge 252, 340, 466, 470, 471
 SMI# 340, 352

Index

SMIACK# 340, 470
Smith algorithm 112
SMM enable 481
SMM handler 340
SMM memory 503
SMM RAM control 503
SMM range 485
SMMEM# 252, 340, 470
SMP 13, 202
SMP OS 13, 59, 60
SMRAM 503
snarf 291, 343, 466, 471, 472, 473, 477
snoop 15, 136, 166, 355, 432, 498
snoop agent 188, 195, 242, 257, 260
snoop phase 192, 195, 197, 257
snoop phase, non-memory transactions 268
snoop ports 135, 142, 170
snoop result 170, 260, 262, 263, 273
snoop stall 260, 267, 268, 503
snooper 15
Soft reset 351
software features register 52
software interrupt 413
software interrupt instruction 410
special 268
special transaction 18, 244, 247, 252, 281, 334, 434, 470, 471, 498
speculative code execution 116, 118
speculative execution 32
speculative reads 120, 124, 125, 126
SPLCK# 227, 253, 316
split lock 227, 253, 434
squashed 136
SS 53
stack 53, 404, 452
stall 117
stall, partial 117
stalled 232, 233
stalled/throttled/free indicator 231, 233
stalling 157
startup inter-processor interrupt 58
startup IPI 14, 60
startup message 56
static branch prediction 85, 110, 116
static branch predictor 113
status 479, 489, 500
STI 404, 405, 409, 410, 413
Stop Clock 355
stop clock 340
Stop Grant Acknowledge 355
stop grant acknowledge 252, 340, 467, 471

Stop Grant state 355
stop grant state 49
store 104, 106, 118, 155, 159
store address unit 107
store data unit 107
store forwarding 104
STPCLK# 49, 340, 438
stubs 182
SUB 225
subordinate bus number 480
subsystem ID 479, 489, 501
subsystem vendor ID 479, 489, 501
superscalar processor 29
switch statement 116
symmetric agent 209, 214
symmetric arbitration ID 49
symmetric request agent 191
symmetric system 202
symmetrical multi-processing 13, 59
sync 252, 339, 466
synchronizing 225, 226
syndrome 435
system error reporting command 492
system error status 492
System Management Interrupt 352
system management interrupt 340
system management memory 494

T

target 188, 195, 242
target abort 273, 313, 317, 319, 323, 326, 498
target ready 278
task gate descriptor 117
task state segment 117, 225, 453
task switch 117, 395, 398, 453
TCK 350
TDI 350
TDO 350
temperature 355
termination voltage 180
TESTHI 355
TESTLO 355
THERMTRIP# 355
throttle 231
throttled 232
time stamp counter 54, 362, 366, 370, 374, 437, 438
time stamp disable 376
timeslice 272, 309
Timestamp Counter 355
TLB 140, 146, 361, 364, 380, 381, 392, 395, 398, 429
TLB error 416
TLB errors 431

TLB, 4MB page 381
TMS 350
toggle mode transfer order 166, 171, 172, 280
toggle-mode transfer order 90
top of system memory 479
top-of-stack 452
TOS 452
tracking 199
transaction deferral 27, 28, 189, 307, 475
transaction ID 191, 252, 273, 316
transaction pipelining 27
transaction queue 463
transaction tracking 199, 230
transfer length 287, 330
translation lookaside buffer 53, 398
TRC 466, 470, 471, 473, 481, 498, 504
TRDY# 278, 289, 321, 343
triple-fault 339
tri-static 41, 47, 354
TRST 350
TSC 54, 362, 366, 370, 374, 438
TSD 376
TSS 117, 225, 410, 412, 453
turbo reset control 470, 471, 473, 504
two-level, adaptive dynamic branch prediction 112
type 0 PCI configuration transaction 18
type 1 PCI configuration transaction 18

U

U/S bit 390, 392, 395
UC 54, 107, 123, 124, 130, 222, 251, 425, 508
UD2 371
uncacheable 124, 251
uncorrectable ECC error 435
uncorrectable error address 492
uncorrectable memory data error 346
uncorrected 425
uncorrected error 425
underflow 448
unified L2 cache 34, 135, 162, 365
unit mask 441
unlocked transactions 274
unpacked data 447
uops 30, 67
UP# 352
Upgrade Present 352
upgrade processor 352
USB controller 498
user bit 441
user/supervisor bit 390, 392, 395
USR 441

V

V bit 513
VAL 425
Vcc5 353
VccP 184, 353
VccS 353
VCNT 507
vector 402
vendor ID 478, 489, 499
vendor ID string 361
version information 361
VID[3:0] 353
video buffer area enable 481
video buffer region enable 490
video frame buffer 380
VIF 407, 413
VIP 407, 410
virtual 8086 mode 403
virtual 8086 mode extension 376
virtual IF bit 405
virtual interrupt flag 407
virtual interrupt pending 407
virtual machine monitor 404
VM bit 406, 409
VM86 403
VM86 mode extension 362
VM86 task 403
VME 376
VMM 404
VMM program 409
voltage divider 180
Voltage ID 353
voltage ID 353
Vref 183, 184
VREF[7:0] 353
Vss 353
Vtt 180, 183, 184

W

wait states 189, 279, 298
WAIT/FWAIT instruction 348
ways 140, 146, 164
WB 93, 107, 124, 127, 131, 135, 149, 227, 251
WBINVD 117, 339
WC 107, 124, 130, 222, 251, 507
WC buffer 107
WCB 125
WP 107, 124, 126, 131, 149, 150, 251
wrap-around 348
write 287, 295
write combining buffer 125

Index

write protect 251
write-back 124, 127, 135, 251
write-back buffer 166, 175, 210
write-combining 124, 251
write-combining buffer 434
write-protect 124, 126
write-through 124, 126, 251
WRMSR 117, 121, 123, 362, 371, 439, 442
WT 107, 124, 126, 130, 149, 151, 222, 251

X

XADD 225
XCHG 225
XOR 225

Y

Yeh's algorithm 112

“**[T]**his series of books is truly an important part of my library. ... I would recommend them to anyone doing hardware design or support, as well as to any developers who write low-level system code.”

—Paula Tomlinson, *Windows Developer's Journal*

Pentium® Pro Processor System Architecture describes the hardware and software characteristics of the Pentium Pro processor, the bus protocol it uses to communicate with the system, and the overall machine architecture.

Written for computer hardware and software engineers, this book details the internal architecture of the processor, providing insight into how it translates legacy x86 code into RISC instructions, executes them out-of-order, and then reassembles the result to match the original program flow. In detailing the processor's internal operations, the book reveals why the processor generates various transaction types, and how it watches bus traffic generated by other entities to ensure cache consistency. *Pentium® Pro Processor System Architecture* also covers:

- relationship of the Pentium Pro to other processors, PCI bridges, and memory
- purpose of the MTRR registers
- detailed description of the data, code, and L2 caches
- processor's power-on configuration
- selection of bootstrap processor
- transaction deferral
- instruction and register set enhancements
- paging enhancements
- VM86 mode enhancements
- Machine Check Architecture
- Performance Monitoring and the Time Stamp

- MMX register and instruction set
- overview of the Intel 450KX, 450GX, and 440FX chipsets

If you design or test hardware or software that involves the Pentium Pro processor, *Pentium® Pro Processor System Architecture* is an essential, time-saving tool.

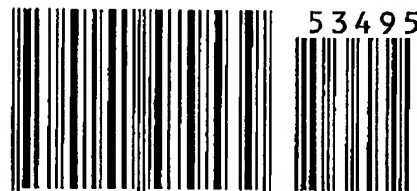
The PC System Architecture Series is a crisply written and comprehensive set of guides to the most important PC hardware standards. Each title is designed to illustrate the relationship between the software and hardware and explains thoroughly the architecture, features, and operations of systems built using one particular type of chip or hardware specification.

MindShare, Inc., is one of the leading technical training companies in the computer industry, providing innovative courses for dozens of companies, including Intel, IBM, and Compaq.

Tom Shanley is one of the world's foremost authorities on PC system architecture and has personally trained thousands of engineers in hardware and software design.

Cover design by Barbara T. Adkinson

Cover photograph by Tazuhiko Shimada/Photonica



9 780201 479539

ISBN 0-201-47953-2

\$34.95 US
\$48.00 CANADA

✧ ADDISON-WESLEY

Addison-Wesley Developers Press

is an imprint of Addison Wesley Longman, Inc.

Find A-W Developers Press on the World Wide Web at

<http://www.aw.com/devpress/>